THE UNIVERSITY OF MICHIGAN

Technical Report ; NR

CONCOMP
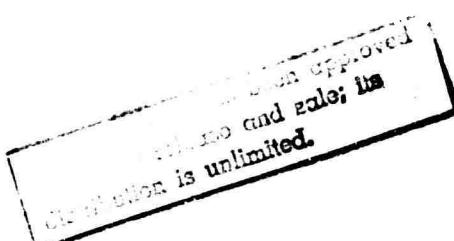
June 1967

TRAMP: A Relational Memory with an Associative Base

William Ash and Edgar Sibley

THE UNIVERSITY OF MICHIGAN

Technical Report 5

TRAMP: A RELATIONAL MEMORY WITH AN
ASSOCIATIVE BASE

William Ash

and

Edgar Sibley

TRAMP: A RELATIONAL MEMORY WITH AN
ASSOCIATIVE BASE

ABSTRACT

This report describes the theory and implementation of an experimental language called TRAMP, which is a software simulation of a content-addressable memory. The system consists of an associative data structure embedded in an interpretive language, allowing great flexibility and strong recursive power. The system has further been extended with a logical inference capability by superimposing a relational structure over the associative memory. The resulting language has already proved to be extremely powerful in several applications, and can be termed a language for developing question-answering and interactive communication systems.

This report discusses the theory and design considerations, details of machine implementation, and details of operation with examples.

TABLE OF CONTENTS

v

LIST OF FIGURES

TRAMP: A RELATIONAL MEMORY WITH AN
ASSOCIATIVE BASE

I. INTRODUCTION

In recent years, the need for a content-addressable
computer memory has become increasingly clear. Larger and
larger programs are being written which require a structured
data base to operate with any efficiency. Many of these could
well benefit by replacing tedious searches with a fast, effi-
cient, "content addressable" access of the data store. A good
example is the "key word" library search. If we ask for a list
of the books written by J.von Neumann, we do not expect the
system to look at each title in its store and save only those
written by von Neumann. And, if there happens to be a catalog
prepared, designed to answer this particular question, we do
not want to have to do a binary search to find the correct
section of the catalog—we want to retrieve the answer directly!

There are many other problems which might find content-
addressability advantageous. Examples abound in Artificial
Intelligence, where prohibitively large tree searches are en-
countered; question answering machines; logical inference systems;
graphic systems; and most conversational (timeshared) systems,
which require immediate, direct access to a large data store to
interact effectively. To date, most investigations into content-
addressable memories have been concerned with hardware; such
memories have not yet proved to be economically feasible. Even
if they had, it is not clear that the obvious gain in speed
would compensate for the loss in generality and flexibility.
For the moment, it can be said that software simulations are a
stopgap measure. They are. But it is not certain that they
will be completely replaced by hardware in even the relatively
distant future.

Another function of an artificial language is to permit
the programmer to phrase his problem in a natural manner. For

many problems information is most naturally described as "relational triples"; e.g., in a graphics system, one might want to say: <Picture in> <Window A> is <Line B>;or<Connected to> <Line B> is <Line C>. The associative processor approach to content addressability allows this.

Before proceeding, we shall explain some potentia ly ambiguous terms.

The essential feature of an associative processor is that it has, in the conventional sense, no explicit addresses. Reference to storage is made by specifying all or any part of an associative cell, and all cells which match this field(s) are referenced. The conventional computer store may be thought of as a special (degenerate) case of an associative memory, in that the association is between the physical address and its contents. However, reference can be made only by specifying the address—one cannot ask directly for all cells which are zero! The true associative memory is accessed by specifying any of the N participants in the association. Associative memories are often referred to as relational data structures. This is because an association between N + 1 "objects" is most easily thought of, talked about, and manipulated by calling it an N-place relation.

The following example demonstrates why an associative processor can effectively be employed as an application of content addressability. Suppose we wish to know the phone number of Clark Kent. It is simple to look it up in the local phone book. It is, however, quite a different matter to find out whose number is 764-6148 (using the same directory). An associative processor would find both tasks equal. In this example, the "association" is between a subscriber's name and his phone number. In translating this to a two-place relation, "phone number of" could be the relation, and using the <R,x,y> format we would say: <Phone number of> <Clark Kent>   is <KR 9-8765>. This is a type of associativity wherein we may

now directly reference this triple by any of its content-address-
able components or combination thereof. If we use only the
first component, phone number, in a search, what will be re-
ferenced is the entire book. If we specify two components: phone
number and 764-6148, then we are referencing directly all assoc-
iations containing those two components, viz., the associations
containing the name(s) of the person(s) having the phone number
764-6148.

    We are, of course, working with a conventional com-
puter memory. The general strategy used to effect the simulation
of an associative processor and an approximation to content
addressability was that of hash-coding.† For those unfamiliar
with the term, hash-coding is simply a technique whereby an
arithmetic transformation is applied to an external name to
generate an internal address. Hash-coding by itself provides a
restricted but significant approximation to content addressa-
bility, but hashing alone does not provide any kind of assoc-
iativy, and there is always the problem of the "collision,"
i.e., when two distinct names hash to the same internal address:
$X \neq Y$; $H(X) = H(Y)$. Hashing partitions the space of names
into equivalence classes. Hopefully, each class has only one
element, but two or more names may be equivalent under this
partition.*

    By providing an interpretive language with an assoc-
iative data structure it is possible to achieve great flexi-
bility. To this end, we decided to use an existing interpreter
and give it a new data structure, rather than start at the bot-
tom by designing a special purpose interpreter. Principally,
we were concerned with the data structure, and the vehicle for

---

† A good survey of this technique may be found in Robert Morris'
article "Scatter Storage Techniques," which appears in the
January 1968 issue of Communications of the ACM, Vol. 11,
No. 1.

* Even restricting names to four characters of the English
alphabet, a one-to-one transformation would require a table
with 456, 976 entries to guarantee no collisions.

it was initially felt to be unimportant, since the data struc-
ture relies on the host only superficially. In considering
the question of the interpreter, e were faced with very little
choice. A major consideration was that of availability; fortunately
this consideration led us to TRAC* T-64 Language. It has
proven to be a most elegant host, and credit for the power of
the resulting system must be shared by both the interpreter
and the associative memory given to it. However, we feel that
the additional primitives are excellent vehicles which change
the original processor into an efficient language for writing
man-machine and machine-machine communication systems. Famil-
iarity with the TRAC language may be helpful in reading this
article, but it is not prerequisite. A brief summary of the
basic components of the language is found in Appendix A.

II. BACKGROUND

An associative processor is one possible tool for
information storage and retrieval, and its history should be
discussed relative to such systems. Unfortunately, adequate
comparisons of different types of data systems are difficult
to make because they are predicated on different rules. Thus
the prime method of storage may vary from cards or paper tape,
through magnetic discs and drums, to prime computer memory;
at the same time, the storage may be either random or ordered
according to some schema; finally, the retrieval of the infor-
mation may be gross, such as the use of a mechanical sort
based on some algorithm, or simple because the data were stored
for such answers or because there is a well-developed language
to address the stored data.

---

* TRAC is the trademark of Rockford Research Institute, Inc.,
  Cambridge, Mass. in connection with their standard languages.
  For details on TRAC T-64 language, see References 14 and 15.

Thus we have three possible criteria for comparison of systems: the type of storage, the way of entering data into that storage, and the language for addressing that storage. The spectrum of potential systems is therefore large and varied. We will consider only those which use main computer memory as the storage device (including virtual or paged memory).

One inherent bias of computer design affects the storage of data: the use of sequential storage, where the data are placed in numbered or ordered cells. Because this organization system allows the automatic <u>indexing</u> of information by means of some automatically varied register, the preferred method of storage is tabular. Fortunately, tables are excellent ways of storing information, for the parallel entries are ways of expressing associations between objects (see Figure 1).

ITEM#	NAME	FATHER OF	BROTHER OF	MOTHER OF
1	JOHN	EDITH; ARNOLD	SAM;JOAN	———
2	ARNOLD	JAMES	EDITH	———
3	MARY	———	———	ARNOLD
4	EDITH	———	———	MELISSA

Figure 1. Association by Tables.

Hence an index may carry the association and we can respond to the query: "Who is the mother of Arnold?" by scanning the "mother of" column, picking the Index 3 at the entry "Arnold" and returning "Mary" from the "3 position" of NAME.

Such retrieval systems are obvious and are in general use. What, then, are the faults? Part of the problem lies in the relative paucity of information—the large number of blanks in the tables. Other problems occur at the time of search, for

the tables are not properly ordered. In fact, the "best order"
depends on what question is expected to be asked. For the
question: "Who is the mother of...?" the order 4,3,2,1 is
preferred, whereas "Who is the father of...?" prefers the
order 2,1,3,4 or 1,2,3,4. Hence there is no order that is
optimal for all questions. Another major problem of tabular
storage is its size limitation. To be efficient, the table
sizes must be pre-specified, and hence a sudden request for
extra space is potentially catastrophic.

The need for easy addition and deletion led to the
list processor, and many information systems stem from the ideas
of IPL-V [16] and SLIP [21]. The former elucidated and refined
the method of pointers and lists, and the ideas of association
lists (Figure 2). The use of lists makes possible dynamically
extended tables. A second use of lists is with association
explicitly defined for certain objects. Thus the illustration
of Figure 2b could be written:

<Attribute A> of <object α> = <value A>
<Son> of <Arnold> is <James>   etc.

Here we see that the question "Who is the mother of Arnold?"
is difficult to answer, because it was not explicitly stored
on Arnold's association list. This question may be answered
by searching all association lists, until one is found which
has the pair {Mother, Arnold}.

SLIP was the first embedding of a list-processing
capability within a higher-level language and was a formative
ring structure. The idea of rings was crystalized by
Sutherland [20], and Roberts [17], and used with data systems
designed primarily for graphics and computer-aided design.
Roberts has also developed a language to refer to rings (Class
Oriented Ring Associative Language: CORAL). In such lan-
guages, the associations are built into the structure by allow-
ing blocks of information to be threaded by rings which carry
the associations between the blocks of data. This is illustrated

List Start

pointer to next

```
           ┌─────────┐        ┌─────────┐        ┌─────────┐
           │         │───────▶│         │───────▶│  stop   │
           ├─────────┤        ├─────────┤        ├─────────┤
           │    A    │        │    B    │        │    C    │
           └─────────┘        └─────────┘        └─────────┘
```

information
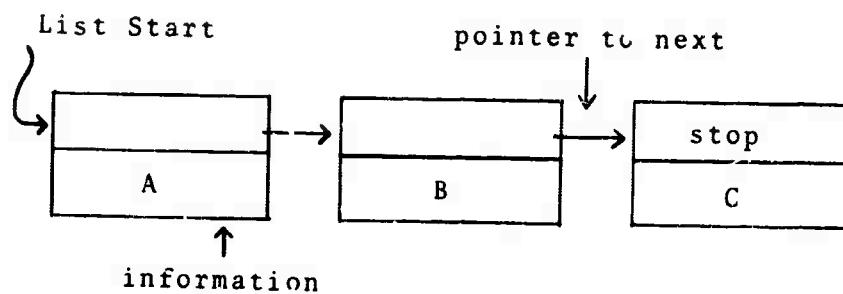
a. Ordinary, single way list: (A,B,C)

Association List of
object "Arnold"

```
      ┌─────────┐        ┌─────────┐        ┌─────────┐
      │         │───────▶│         │───────▶│  Stop   │
      ├─────────┤        ├─────────┤        ├─────────┤
      │         │        │         │        │         │
      └─────────┘        └─────────┘        └─────────┘
           │                  │
           ▼                  │
   ┌─────────┐   ┌─────────────┐          ┌─────────────┐
   │         │──▶│    stop     │          │             │
   ├─────────┤   ├─────────────┤          ├─────────────┤
   │ att SON │   │ value JAMES │          │     ...     │
   └─────────┘   └─────────────┘          └─────────────┘
                        ▼
              ┌─────────────┐   ┌─────────────┐
              │             │──▶│    stop     │
              ├─────────────┤   ├─────────────┤
              │ att SISTE?  │   │ value EDITH │
              └─────────────┘   └─────────────┘
```
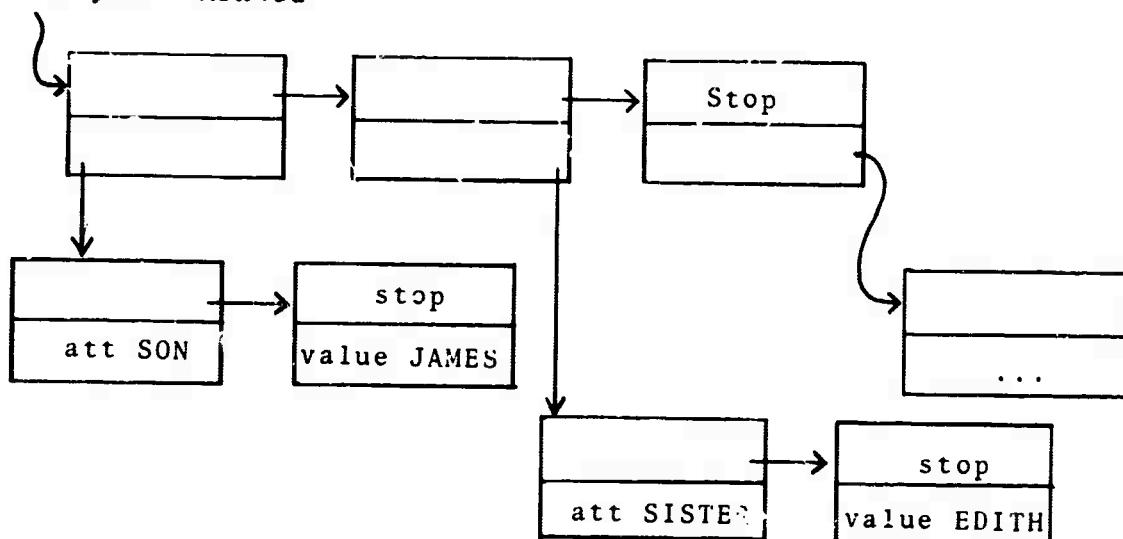
b. One type of paired Association List:
    [(att A,val A)(att B,val B) ...

Figure 2. Association by Lists

by Figure 3. Dodd [1] has implemented a similar structure
within PL/I. The duality of certain relationships, such as:
"defined by" and "defines" or "to the left of" and "to
the right of," etc., led to the need for a connector block,
here illustrated by the three NUBS.* In essence, the NUB
represents a two-way switch for transferring out of one ring
and into another. The subrout nes or macros pass along the
ring until they arrive at a NUB. They "switch" it, and pas
into the other ring, passing along the second ring (and others
as found) picking up information until they return to the
original NUB and re-enter the first ring. This allows
answers to questions such as "Who is the mother of Arnold?"
as well as "Who is the son of Mary?" One of the major dis-
advantages of these structures occurs on adding a new, not
previously anticipated, association. The operation either is
impossible, requiring a complete recompilation, or else clumsy,
patching on additional blocks (Figure 4) and requiring so-
phisticated garbage collectors. A recent survey by Gray [5]
describes these and similar structures.

Probably the first conception of a true relational
data structure was Kochen's AMNIPS [7,8]. He dealt with the
problem of logical inference rather than with data structures,
but he recognized that conventional memories were inadequate,
and turned to the relational structure, which unfortunately
was never fully realized.

A considerable amount of work has been done on
various "question-answering machines." Among the better early
machines were Lindsay's SAD SAM [11] which can digest English
statements about family relationships and construct a family
tree, and the BASEBALL program of Green, et al. [6] which
answers English queries about facts taken from a stylized

---

* It is interesting to note that there is no need for NUBS
  if we are willing to store all inverse and similar relation-
  ships explicitly, with separate rings for each.
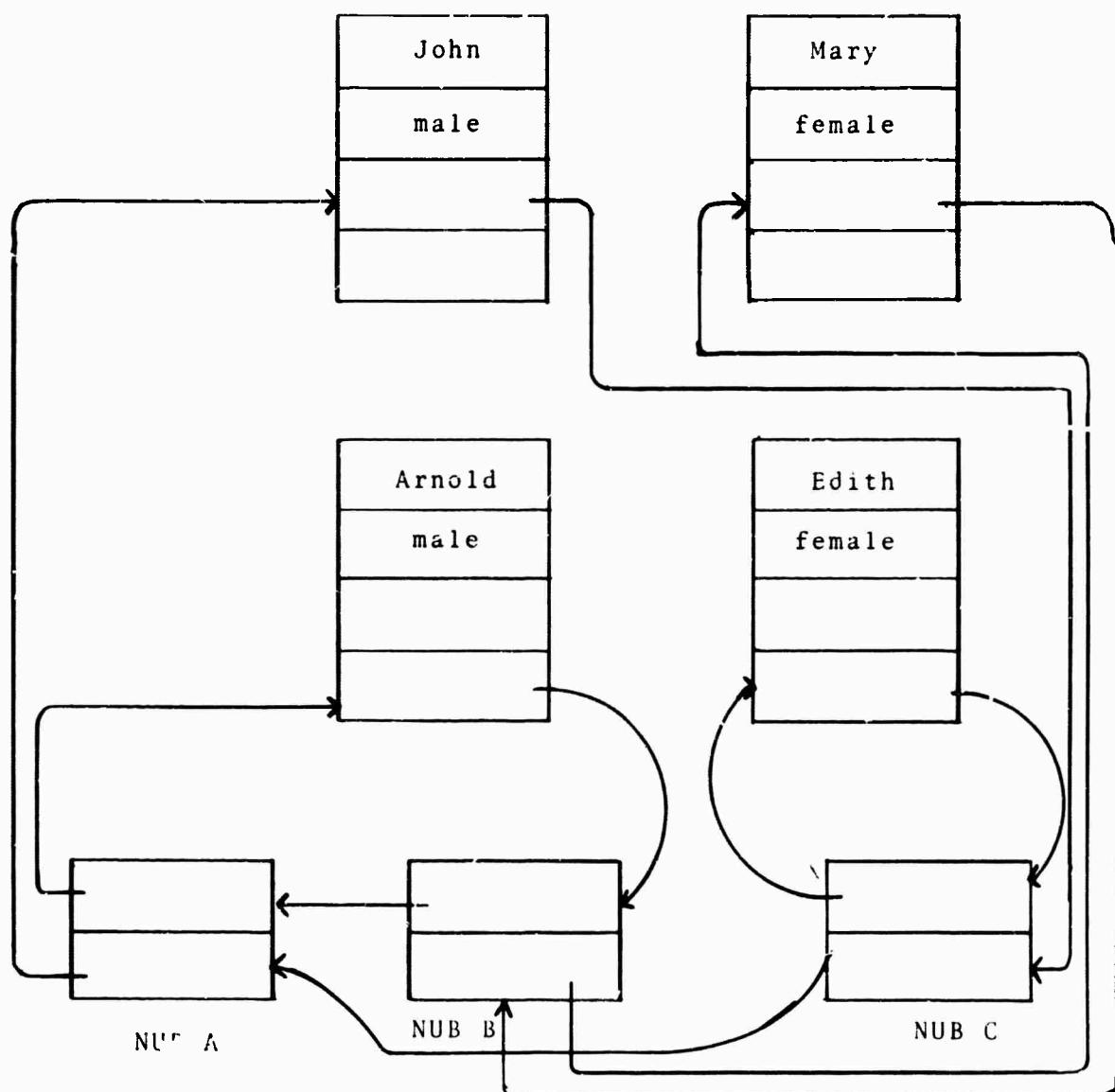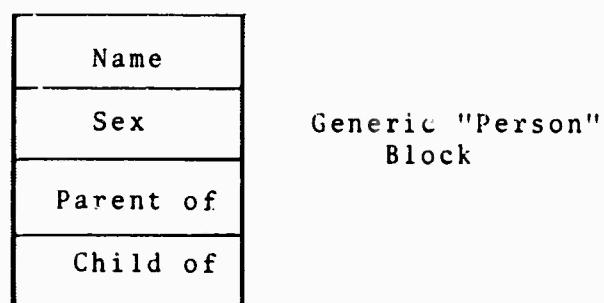
Figure 3. Associations by Rings.
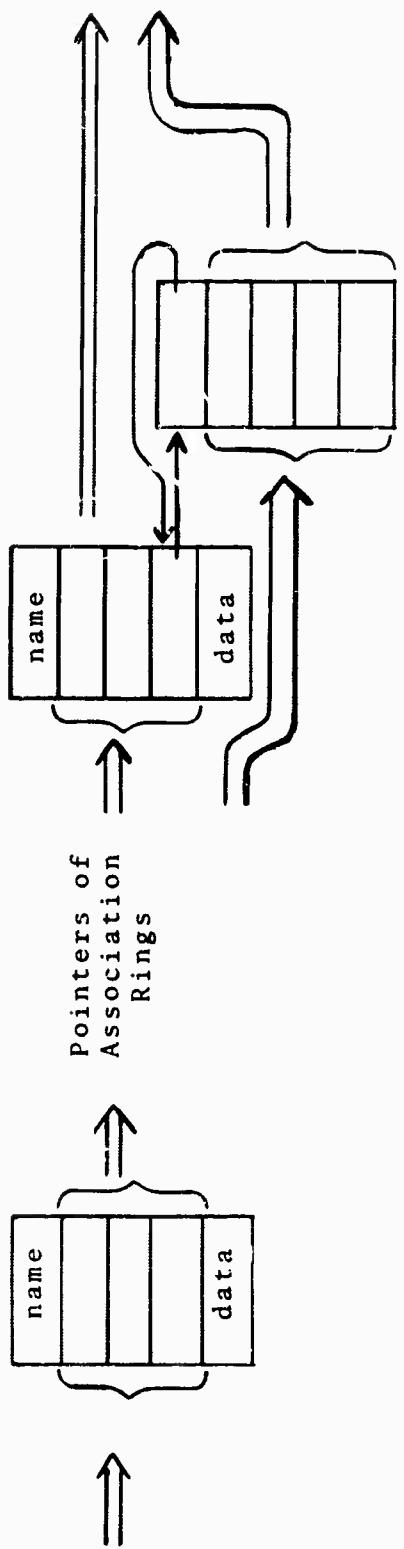
Figure 4. Addition of Association with Rings.

Pointers of
Association
Rings

name

data

name

data

-11-

baseball "yearbook." These investigations use conventional
data structures, and their real contributions were to the
analysis of the English language and logical inference by
computer. Simmons [18] provides the most complete survey to
date of this work.

Maron and Levien [9,10,12] have designed one of the
most extensive and complete systems depending on a relational
data file. They deal with binary relations as their building
blocks, i.e., each association has three components: one
relation and two operands. In addition, they allow the naming
of the entire association (triple) giving rise to a fourth
component. Reference can be made using any of the four com-
ponents, and there are four copies of each association—one
copy for each word which can reference it.

Feldman has recently used the ideas of hash-
coding for association tables [3,4]. The associations
are, of course, carried by a new table, referenced
by a "double hash" technique. Feldman's language, AL , is
designed to be compiled rather than interpreted. AL has been
expanded to be three languages in one: a true ALGOL-type
algebraic language with full numerical computation facilities;
an associative processor; and a language which operates on
sets as its basic entities with a full complement of set
operations. The language now allows for certain kinds of re-
stricted composition of associations. Specifically, if a
sentence is a triple $<A,0,V>$, then 0 or \ may itself
be another triple. This allows for generating N-place rela-
tions out of the basic binary relations. However, the language
has, instead of dynamic inference, a DO loop which is slight-
ly more cumbersome, and far less economical of storage,than
are strict logical inference capabilities.

Naturally, the simplicity in using data systems de-
p.nds on the retrieval language. We have already suggested
that the problem is partially a function of explicit versus
implicit storage. If the family relationships are taken in

the binary male lineage tree of four generations (Figure 5),
we have: 16 great-grandfather—son; 24 grandfather—son; 28
father—son; 14 brother; 24 uncle—nephew; 48 cousins, etc.
Obviously, a large number of relationships must be explicitly
stored for rapid access, compared with the cost of an implied
relationship search with small storage requirement.

The "relational" part of the TRAMP system is the
means for retrieving implied relationships, while the "assoc-
iative" part deals with the explicit relations.

III. TRAMP

TRAMP (Timeshared Relational Associative Memory
Program) is two packages of functions: the first—the data
structure—may be used to enter, retrieve, and generally manip-
ulate an associative data structure; the second—the relational
memory—places an artificial structure on the "associative
triples," viz., the relational structure. The relational
package allows logical inference to be performed on the data
within the associative structure. Specifically, rules may be
entered, these will be followed by TRAMP, effectively expanding
a "minimal" set of data to a workably large set; the number
of associations that must be explicitly stored is thereby
drastically reduced. For example: by defining the relation
"HUSBAND OF" to be the converse of "WIFE OF," the user need
only store marital relations in one direction, while effectively
having them available in both directions. More detailed ex-
amples and the rules for using the relational package appear
later.

These machine-coded functional packages are presently
embedded in the UMIST* interpreter on the IBM/360 model 67.
Although this existing union has proved most fruitful, the

---

* UMIST is closely patterned after the standard TRAC T-64
Language, and was implemented at The University of Michigan
with the cooperation of Mr. C.N.Mooers, creator of the TRAC
T-64 language.
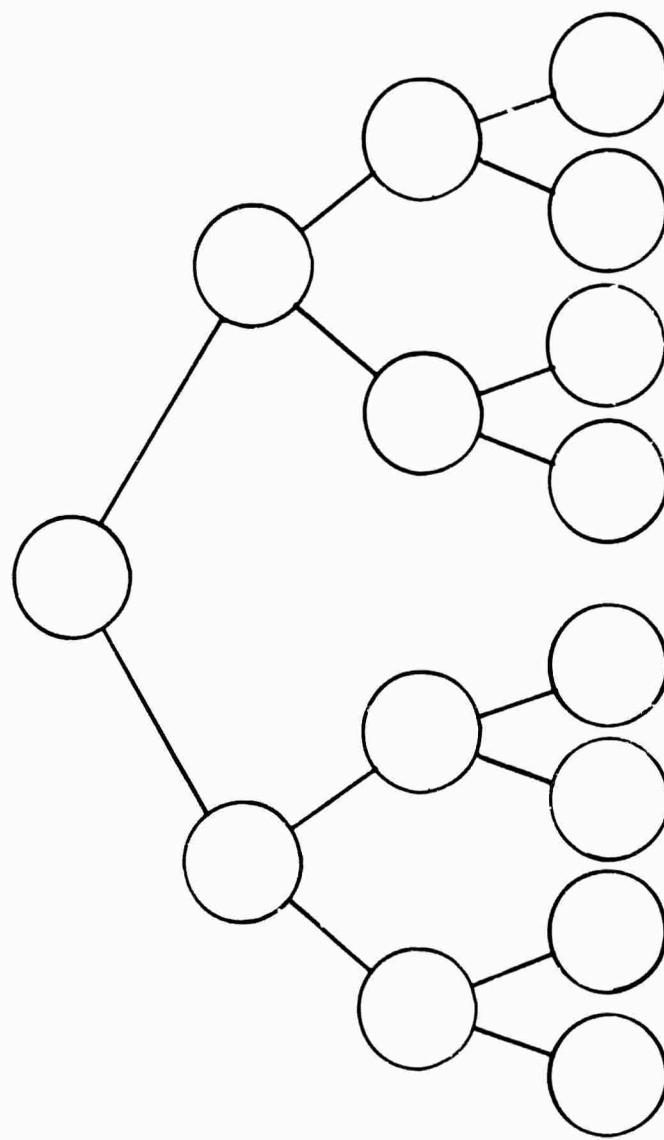
Figure 5.   Binary 4 Generation Male Relational Tree

data structure is totally independent of the interpreter and
actually relies on it only for I/O. The relational package
is also independent, except that it relies on the type of re-
cursion that the interpreter provides. The relational package
is totally dependent on the associative data structure.

The Associative Data Structure

Feldman's initial work was a strong motivation in
the design of this system, and led us to adopt his notation,
viz., the generic entity:

$$A (0) = V$$

<u>A</u>ttribute> of <<u>O</u>bject> equals <<u>V</u>alue>

Thus the <u>Associative Triple</u> is: <A,0,V> . Each of the three
components is a non-empty set. To the data structure this is
an ordered triple, but no interpretation or meaning is attached
to the ordering, and all three are treated equally, giving none
a priority.* By appropriately designating the three components
as being constant or variable, we can ask eight "questions"
of the data structure. Again using Feldman's notation, with
a slight re-ordering, they are:

F0	A (0)	= V
F1	A (0)	= x
F2	A (x)	= V
F3	A (x)	= y
F4	x (0)	= V
F5	x (0)	= y
F6	x (y)	= V
F7	x (y)	= z

---

* This is in contrast to the relational package which places
an artificial structure on the triple, i.e., calling the
first component a "relation" and the second and third its
arguments.

where [A,0,V] represent constants, and [x,y,z] are variables.
Question F7 is not a question at all but a request for a dump
of the associative memory, and in TRAMP such a dump is given.
Question F0 simply asks: "Does A (0) = V?" and the answer is
a kind of truth value. In the case where A, 0, and V are
all singletons, the truth value is a straightforward 1 or 0 de-
noting whether or not the specified association can be verified
by the data. The interpretation is slightly ambiguous, however,
when one or more of the three sets has cardinality greater than
one. To illustrate, assuming that the association

COLOR (CAR) = RED; GREEN*

has been stored, these five questions have the following truth
values:

1.	COLOR (CAR) = BLUE	0
2.	COLOR (CAR) = RED; GREEN	1
3.	COLOR (CAR) = RED; BLUE	?
4.	COLOR (CAR) = RED	1
5.	COLOR (CAR) = RED;GREEN; BLUE	?

Questions 1 and 2 are clearly false and true respectively,
but questions 3 and 5 are each partially true and partially
false; question 4 is only half true. The interpretation which
seemed most natural, and the one adopted by TRAMP, gives the
truth values as shown, namely:

if ALL associations implied by the question are
resident in memory, or derivable therefrom, the
value is "1"

if none, the value is "0"

if some, but not all, the value returned is "?"

* Since the comma already plays an important role as a TRAC
language meta character, it is unavailable as a set element
delimiter. Therefore the semi-colon (;) plays that role
in TRAMP.

Questions F1-F6 simply ask the system to "fill in the blank(s),"
i.e., to replace the variable with the set that is the answer to
the question. For example, Question F1 asks for the set of all
Vs that A (0) equals. Question F3 asks for the sets of all
Os and Vs that have a first component "A." Because of the
recursive nature of TRAC, questions F1-F6 may be nested in
any way, to any desired depth. One may ask: "How many fingers
on a hand?"; "What figures are pointed to by the arrows in
Window Q?"; "How old are the fathers of the wives of Mary's
brothers?"; or any questions composed in any way compatible
with the stored data, nested to any level.

For those totally unfamiliar with TRAC language, for
this section it is necessary to know only the syntax of a func-
tion call. The sharp sign (#) signals the start of a func-
tion call, with the call itself enclosed in an immediately
following pair of parentheses. The arguments are separated
by commas, and the first argument is the name of the function.

#(sub,ARG)

is therefore analagous to the FORTRAN

CALL   SUB(ARG)

Data Structure Storage

The name of the storage function is dr and the
syntax of the call is: #(dr,A,O,V) . Again, the three ar-
guments to dr are each non-empty sets. Each point in the
cartesian product of the three sets is stored, i.e., each
element of each set is grouped with each pair of elements of
the other two sets, and the resulting triple is stored. Thus
a single call on dr stores as many associations as the pro-
duct of the cardinalities of the three sets. The storage
declaration:

#(dr,AGE,JOHN;MARY,64)

would therefore store:

<p style="text-align:center;">AGE (JOHN) = 64</p>

<p style="text-align:center;">AGE (MARY) = 64</p>

The actual storage is accomplished by pairing each A and O to point to a list of Vs ; each A and V point to a list of Os , etc. These "answer" lists are, strictly speaking, un-ordered, except that they retain the order in which they were stored. That is, asking the question:

<p style="text-align:center;">"Whose age is 64?"</p>

would yield the answer:

<p style="text-align:center;">JOHN; MARY <u>not</u> MARY; JOHN</p>

It should be noted that this is a pure data structure, and it does not deal with semantics; <u>dr</u> simply inserts associa-tions into memory in a way that they can be quickly retrieved. TRAMP is not a question-answering system that checks for redun-dancies or inconsistencies of data.

Data Retrieval

The primary retrieval function has the name <u>rl</u> . The syntax of the function call is identical to that of <u>dr</u> except for variable specification. A variable in TRAMP is denoted by enclosing a name, possibly null, within asterisks (*) . Thus, #(rl,A,O,V) has no variables and asks whether A (O) = V ; #(rl,A,O,*X*) asks: "What does A (O) equal?" If the vari-able is <u>Named</u>, i.e., there is a name within the asterisks, then the function is <u>Null Valued</u> and the answer is stored in TRAC language form storage labeled by the Name. #(rl,A,O,*ANS*) would store the set of Vs which A (O) equals, under the label "ANS." If the variable is not Named, #(rl,A,O,**) , then the answer is the <u>Value</u> of the function. #(rl,A,*SETO*,**)

: an example of a two-variable question with one Named and
on Unnamed variable. The result in this case would be that
the set of Os is placed in form storage under the label
"SETO" while the set of Vs is returned as the Value of the
function.

The two-variable questions (F3, F5, F6) simply use
the Name table of one of the variables and index through that
table, internally always asking the one-variable questions.
Since the data structure does not assign any priority to the
three components, questions F3, F5, and F6, although con-
siderably slower that the one-variable questions, are all
equal among themselves. The process of answering a two-vari-
able question is less efficient because it must iterate on
the one-variable questions, the number of iterations being a
linear function of the size of memory.* The speed with which
the one-variable questions are answered is not significantly
affected by the size of memory! The three-variable question,
#(rl,**,**,**) , is, of course, the slowest of all and it is
a full dump of the associative memory. Alternatively, one can
call: #(dump).

Going back to the earlier example of N and MARY,
the question: #(rl,AGE,JOHN;MARY,**) should have as its
value: 64;64. That is, redundancies can be valid and should
be reported. But there are certain times, particularly in
the two-variable questions, when redundancies become quite a
nuisance (and even threaten to overflow the interpreter).
Therefore, the function rl will always return an answer set
with all redundancies deleted. A second entry point is pro-
vided, with the name rlr , which is identical except that it
does not check for redundancies but returns the answer set
as it finds it.

---

* Here, and subsequently, "size of memory" refers to the amount
  of data in the structure, rather than the physical extent of
  the system.

rl generates the union of the answer sets. That is, the question: #(rl,AGE,JOHN;MARY,**) has two answer sets: the AGE of JOHN and the AGE of MARY. rl simply forms the union of however many sets there might be. int is the function (yet another entry point to the same routine) which generates the intersection of the several answer sets. Thus, #(int,SOUTH;WEST,AUGUSTA,**) generates the set of all things both south and west of Augusta. #(rl,SOUTH;WEST, AUGUSTA,**) , on the other hand, would generate the set of all things either south or west of Augusta.

Data Structure—General Strategy

As stated in the introduction, hash-coding is the technique most basic to the data structure design. A brief description of the use of hash-coding in TRAMP follows. Section V gives a more technical and detailed description.

The data structure uses three Name tables and three Association tables, one each for each of the three components of the association. When the declaration: #(dr,WIFE,JOHN, MARY) is made, each name that appears must be stored somewhere in memory. The full name must be present so that it can be retrieved, and so that, when it is referenced, a collision can be identified and resolved. The first hash, H_1 , then is applied to the "A" component, "WIFE," to generate a displacement from the A Name Table. The designated table entry is then inspected. If the entry is zero, then there is no collision and WIFE has never appeared before as an "A" component. Accordingly, the table entry is now made to point to the Header for the name WIFE, see Figure 6. If the table entry is not zero, the Header to which it points is inspected to see if it is the Header for WIFE. If so, the A name has been processed and we move on to the O component, otherwise there is a collision. For a collision, instead of a single Header, there is an alphabetical list of Headers.

H$_1$(name)

name table

count of associations name is used in

alphabetical

T H CT next

pointer of head of sublist

first 6 characters next

T H CT next

T H CT stop

sublist as long as EBCDIC representation requires

stop

pointer of tail of sublist
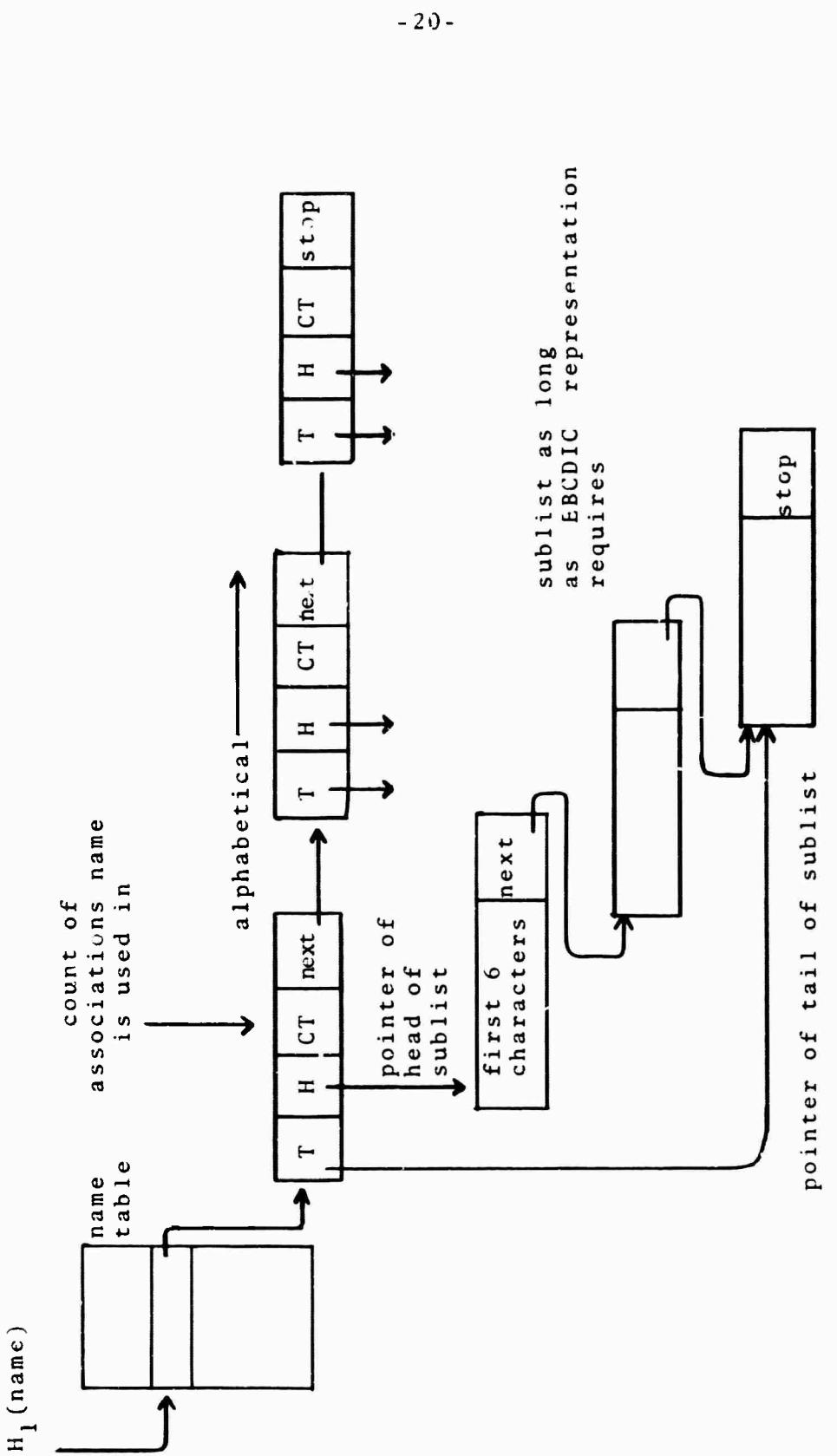
Figure 6. Name Table Structure

Thus, Name Table collisions are not really special cases: if there is no collision, then there is a list consisting of a single Header, otherwise the list contains two or more Headers in alphabetical order.

If the above process did not find the name, before it is actually placed in storage, a further check is made on the other Name Tables, thus avoiding redundant storage. Any name will appear at most once in memory, with up to three Headers pointing to it.

The same procedure is applied to "JOHN" and "MARY," the O and V of this example, on their respective Name Tables. As a result of the Name Table processing, a unique pointer is associated with each of A, O , and V , namely the pointer in the Header which points to the location of the actual name. It is this unique pointer that will be used for the second hash, H_2 . "WIFE" must now be placed on the <u>A Association Table</u>. To do this, the O and V pointers are hashed together to generate a displacement from the Association Table. To be able to identify collisions, both pointers that were used to generate the hash are stored in the table entry designated by the hash. Collisions are again resolved by ordered lists. The Association Table entry has three pointers: the first two are the pointers used to generate the displacement; the third points to the <u>Answer List</u>, i.e., the list of A's (in this case) with which O and V have been associated. Thus, "WIFE" is appended to the Answer List by placing the unique pointer to it at the end of the list. Note that H_1 is a function of the actual name, while H_2 is only a function of where the name is stored and is independent of the name itself. Figure 7 shows the Association Tables, both for the collision case [7b], and for the normal case [7a].

-22-
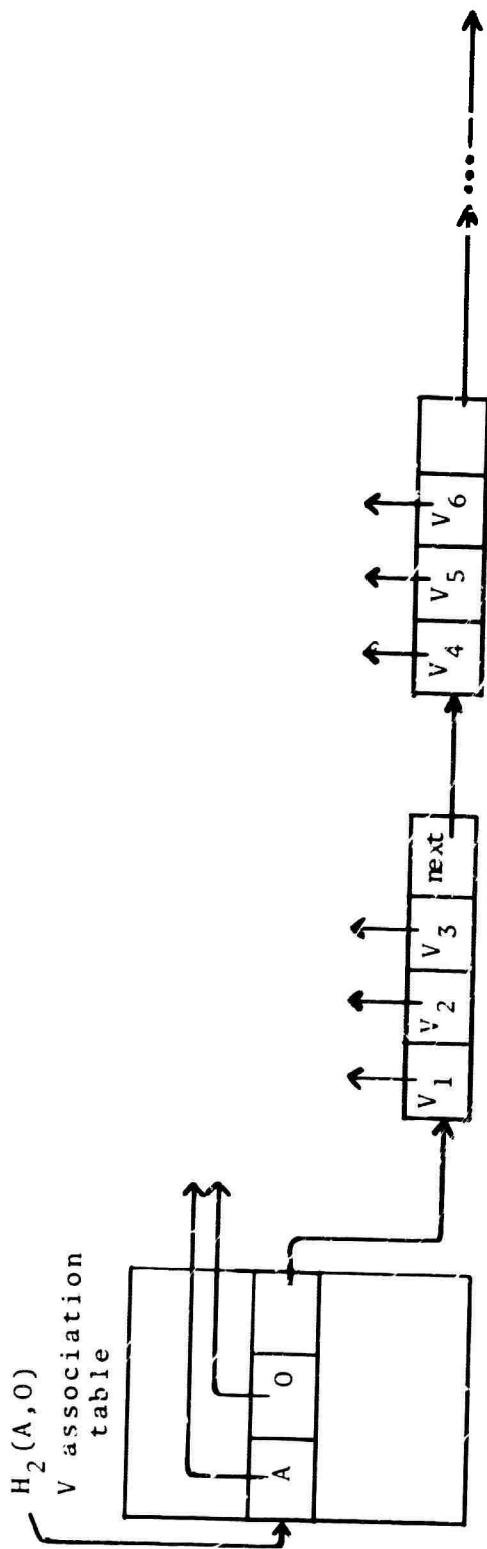
Figure 7a. Normal Case: No Collision.
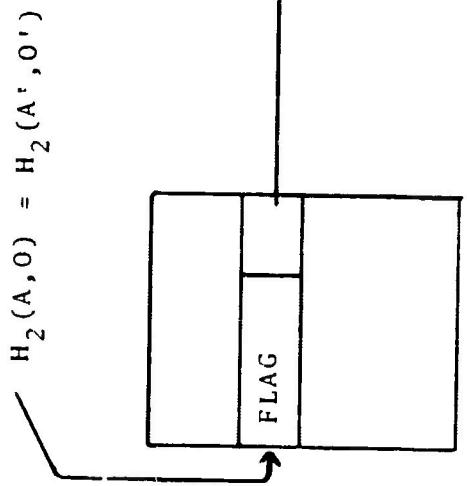
Figure 7b. Collision.

FIGURE 7. ASSOCIATED TABLES.

IV. LOGICAL INFERENCE PACKAGE

The associative memory accomplishes a kind of content addressability by using two quick hashes to address data, and the access time is essentially independent of the size of storage.* But for most, if not all, applications, many associations will be implied by a single associative sentence. This poses two real problems:

1. The user must make sure that all associations that apply are actually inserted into the structure. This is extremely tedious and prone to error and omissions.

2. Explicit storage results in gross inefficiency.

To alleviate this, TRAMP provides the facility to define, in a characteristic way, what other associations may be derived from a given association. This permits all of the information that might be contained in a single association or sequence of associations to be utilized instead of having to enter the same information redundantly in each of the several ways that it might be referenced. The name of the function which makes the definition is <u>ddr</u> . The syntax of the function call is: #(ddr,(R = EXP)) , where R is the relation ("A" component) to be defined, and "EXP" is a logical expression which is the definition.

Before presenting examples of the use of <u>ddr</u> , two relational operators must be defined:

The first is <u>converse</u>, denoted in TRAMP by ".CON." Converse simply inverts the order of the two rela-

---

* As the size of storage increases, there are more collisions, but they are quickly resolved, and do not cause a significant delay. Even in extreme pathological cases, they involve only relatively minor list searches.

-23-

tional arguments:†

$$R(x,y) \leftrightarrow .CON. \; R(y,x)$$

Thus "CHILD OF" is the converse of "PARENT OF"; "WIFE OF" is the converse of "HUSBAND OF"; "SPOUSE" is its own converse; any symmetric relation is its own converse.

Relative Product: The relative product or composition of two relations is commonly denoted by R_1/R_2 , and this is the notation used by TRAMP.

$$\forall x \forall y \exists z \; [(R_1/R_2)(x,y) \leftrightarrow R_1(x,z) \wedge R_2(z,y)]$$

Less rigorously, but more specifically,

$$\#(ddr,(R_3 = R_1/R_2))$$

would tell TRAMP that $R_3(x,y)$ if a "z" can be found such that $R_1(x,z)$ and $R_2(z,y)$.

Besides these two relational operators, three logical operators are available: .A. (conjunction); .V. (disjunction); .N. (negation). Finally, there are six equality operators: .EQ.; .NE.; .GE.; .LE.; .GT.; .LT. , with obvious meanings.

Examples of TRAMP relational definitions are:

  #(ddr,(BIGGER = BIGGER / BIGGER)) Bigger is transitive

  #(ddr,(BIGGER(A,B) = BIGGER(A,Q) .A. BIGGER(Q,B)))
                  exact same definition using expanded
                  format—specifying dummy arguments.

  #(ddr,(SIB = BRO .V. SIS V. .CON.SIB))
                  a sibling is a brother or a sister and it
                  is symmetric.

† The relational notation used by TRAMP is derived from the format "R,x,y" by enclosing the relational arguments in parentheses. This is a slight distortion of the associative notation: A(0) = V , but the order is preserved: R(x,y) means that R(x) = y .

```
#(ddr,(HUSBAND = .CON.WIFE))    Husband is the converse of Wife.

#(ddr,(BIGGER = LARGER)    Bigger and Larger are synonomous.

#(ddr,(BRO(CAIN,ABEL) = SIB(CAIN,ABEL) .A. SEX(ABEL,"MALE")))
                         a brother is a male sibling.  Note that
                         constants are denoted by enclosing them
                         within double quotes.

#(ddr,(MALE(X) = SEX(X,"MALE")))    defined the unary relation
                                    MALE

#(ddr,(BRO(X,Y) = FATHER(X,Z) .A. FATHER(Y,Z) .A. MALE(Y)
                                              .A. X.NE.Y))
                     a brother is a male offspring of the same
                     father, other than oneself.

#(ddr,(STEPMOTHER = FATHER / SPOUSE .A. .N.MOTHER))
                     a stepmother is the spouse of the father
                     who is not the mother.

#(ddr,(NEPHEW = SIBLING / SON))    a nephew is the composition
                     of sibling and son.

#(ddr,(UNCLE = .CON.(SIBLING/SON)))    in a male world, uncle
                     is the converse of nephew and may be de-
                     fined as the converse of the definition
                     of nephew.

#(ddr,(UNCLE = .CON.NEPHEW))    or simply as the converse of
                     nephew.
```

More complex examples will arise, and TRAMP is pre-
pared to handle definitions of the above form to a level of com-
plexity virtually unlimited. One major constraint is placed on the
definitions: relations must be defined so that at least one
set is <u>generated.</u> This generated set can then be intersected
with or joined with another set, or otherwise manipulated.
The intent of this constraint is that there be at least one
<u>reference</u> set. The "whole space" may never be used as a
reference set.

```
#(ddr,(R₃ = .N.R₁ .V. R₂))    is illegal since it specifies
                              a global complement  (of  R₁),
                              i.e., it references the "whole
                              space."
```

$\#(ddr, (R_3 = .N.R_1 .A. R_2))$ is legal because it specifies a relative rather than a global complement, i.e., R_1 places a constraint on R_2 not on the whole space.

Implementation of Inference

The purpose of the inference mechanism is to allow the user to define under what conditions an implied association may be derived from data explicitly in memory. This is accomplished by generating where necessary (where defined) a more complex retrieval call from a simple one. Specifically, if the following definition had been entered:

$\#(ddr, (STEPMOTHER = FATHER/SPOUSE .A. .N.MOTHER))$

then the following simple retr₁ al call:

$\#(r1, STEPMOTHER, JOHN, **)$

which asks for the stepmother of John, would be expanded by the system to be the following:

$\#(rcom, \#(r1, SPOUSE, \#(r1, FATHER, JOHN, **), **), \#(r1, MOTHER, JOHN, **))$

$\#(r1, STEPMOTHER, JOHN, **)$

The exact call generated would be slightly different, but that is a technicality, irrelevant at this point. The final retrieval call in the sequence generated asks if the desired association was entered explicitly. It is always assumed that a relation that has been given a definition may also appear explicitly. The rest of the expanded call will find the answer if it is present implicitly. This expanded call is then returned to the UMIST processor, which in turn makes the actual calls to the data structure. The importance of this is that relations need be expanded only one level at a time, with the UMIST recursion automatically taking care of the possibility that any relation is defined in terms of more complex relations,

etc. (this is the major difference between the call as it actually would be generated, and as it appears above—the above, taken literally, would specify an infinite recursion!) Thus the inference compiler generates TRAMP procedures--they operate only within the TRAMP language—not at a lower, machine level. The definition, entered by #(ddr), specifies what information the procedure is to derive and what rules may be used to derive it; the compiler accordingly constructs such a procedure; and the interpreter (TRAMP inference interpreter—rather than UMIST) expands the procedure at retrieval time, filling in information specific to the call.

At retrieval time, a retrieval "preprocessor" looks to see if the "relation" ("A" component) has been given a definition. If not, the preprocessor exits and retrieval proceeds as described earlier. If the name is found to have been defined, then the "interpreter" is called in to interpret the program generated by the compiler at the time it was defined. This program tells the interpreter what TRAMP function calls are to be made, and what the function arguments are to be.

It should be noted that the compiler actually puts out two programs: one which, given x of R(x,y) , builds a chain to generate y ; the other builds the appropriate chain in the opposite direction, from y to x . Thus question F1: #(r1,A,O,**) generates a different sequence of function calls than F2: #(r1,A,**,V) . It may not be immediately obvious why this is necessary, but, in general, the two programs will be quite different. This is always the case for composition. Still, the compiler would only have to output one program, and the interpreter could decide how to interpret it. Since the compiler will usually be called only once or twice for each relation, or certainly fewer times than the interpreter, it is most efficient to let the compiler do as much of the work as possible.

The compiler is prepared to handle definitions which are circular in the sense that a relation is defined in terms of itself. That is, symmetric and transitive relations are perfectly acceptable. However, the sequence:

#(ddr,(PPP = QQQ .V. ...))   #(ddr,(QQQ = PPP .V. ...))

is invalid because of its circularity. Were the compiler to
attempt to generate code for that sequence, the code would
specify an infinite recursion. This situation is checked
for and flagged if detected.

V. TRAMP INTERNAL ORGANIZATION

In effect, TRAMP employs a triple storage technique
to be able to reference an association in three different ways.
Thus, the association A (0) = V is stored on each of the A ,
0 , and V Association Tables. This makes the answers to
questions F1, F2, and F4 equally accessible and optimizes
retrieval time.

TRAMP uses eight principal blocks of core. Though
it is designed to run under a timesharing supervisor which
continually swaps TRAMP on and off a drum, TRAMP itself makes
no explicit use of drums, discs, or other secondary storage
devices; that is, such use by the system is transparent to
TRAMP as well as to the user. The blocks of virtual core are:
four Name Tables [A, 0, V, and a Name Table for Defined Re-
lations]; three Association Tables; and a General Storage list
[GS] (commonly called "Available Space" in many list proces-
sors). GS provides all the working space for TRAMP and is
by far the largest of the eight. In addition to storing all
of the information indexed by the seven tables, GS resolves
any overflow (via collisions) from the tables.

For purposes of illustration, let us follow the
interpretation of:

#(dr,HUSBAND,EVE,ADAM)

First the Name Tables are processed. "HUSBAND" is hashed to
produce a displacement from the "A" Name Table. The actual

hashing scheme for H_1 is to form a full word (4 bytes) by con-
catenating the first, last, and middle two characters of the
name, in that order. A single character may play only one of
those roles, i.e., a name consisting of one character has no
last or middle characters. Any missing components are filled
with hexadecimal zeroes. Thus HUSBAND yields "HDBA"; EVE yields
"EVo" and ADAM yields "AMDA." The full word so generated is
then transformed, with the transformation being little more
than squaring and masking.

A list is generated in GS to hold the EBCDIC re-
presentation of the name: 6 characters (bytes) per double word
and a 2-byte pointer to the next unit in the list. All units
in GS are double words (64 bits). Each name list is term-
inated with a stop meta character. All lists in TRAMP are
terminated with a stop pointer, though the stop pointer is
superfluous in the name lists because of the stop meta. In
the case of HUSBAND, two double word units will be needed: the
first will hold the 6 characters H-U-S-B-A-N, and a two-
byte pointer to the next unit which will hold the character
"D," the stop meta character, and the stop pointer, with four
bytes left over. Since HUSBAND is used here as an "attribute,"
we are concerned with the A Name Table. We look at the
entry in this table designated by the hash. If this entry is
empty, i.e., zero, there is no collision, and HUSBAND has
not been used previously as an attribute. If not empty, we
look at the list of Headers pointed to by the entry. This
list is alphabetical, so we need look only until we find HUSBAND
or its proper alphabetical position on the list. Proceeding
down this list of Headers, we compare the list generated above
with the sublists pointed to by the headers. If a match is
found, simply return this temporary list to GS and increment
the USE count for HUSBAND in its header. If it is not found,
insert it ... after checking the O and V tables for its
occurrence. Previously HUSBAND may have been used as, say, an

"object" #(di,SEX,HUSBAND,MALE) . In this case the HUSBAND
name list would already be resident in GS. We therefore return
the copy of it generated above, and insert a pointer to the
first list on the A table header list. Thus a name never ap-
pears in core more than once, though many pointers may point
to it, including up to four headers if a name appears on all
four Name Tables.

The above process is done for each HUSBAND, EVE, and
ADAM. The final pointer to the one name sublist of each is
saved to generate the Association Table hash$-H_2$. Let us
follow the processing of the O Association Table. HUSBAND and
ADAM (A and V) are hashed together (multiplied and masked)
to produce a displacement in the Association Table. The actual
hash is performed on the two unique pointers found during Name
Table processing. The designated Association Table entry is
examined. If zero, there is no collision, and HUSBAND and
ADAM have never appeared together with another value. The
unique pointer to HUSBAND is placed in the first 2 bytes of
the 6-byte entry. The pointer to ADAM is inserted in the middle
2 bytes. A double word unit is picked off GS to be the start
of the "Answer List," and the pointer to this Answer List is
placed in the last 2 bytes of the table entry. The Answer List
elements consist of three 2-byte pointers to name sublists, the
answers, and a 2-byte pointer to the next list element. Ac-
cordingly, the pointer to EVE is inserted in the Answer List.

If the table entry was not zero, compare the first 4
bytes of it with the two pointers that would go there. If a
match is made, then just add EVE to the end of the already
started Answer List (polygamy is fine here), starting a new
unit if necessary. This is a simple unordered list, with new
elements always being added to the right-hand end. If the first
four bytes of the table entry do not match the pointers, is
there a collision flag in the entry? If not, a collision list
is begun. This is an ordered list, with the ordering being

the numerical value of the full word obtained by concatenation
of the A and V pointers (the first 4 bytes of the table
entry). Each element of the list is a double word as always.
The first 6 bytes of this double word are identical to the
6-byte table entry. The last 2 bytes point to the next element
on the list. When a collision occurs, the first 4 bytes of
the table entry are so flagged and the last two bytes point to
the list which resolves it.

The entries of all the tables, as well as all list
pointers within GS , point to double word units in GS . All
pointers are 2 bytes long (16 bits), but are capable of ad-
dressing 128 pages of GS . (1 page = 4096 bytes; 128 pages =
2^{19} bytes.) For some applications this size is more than
adequate,and for others (e.g., artificial intelligence) not
nearly enough. With its present scheme (addressing 2^{19} bytes
with only 16 bits) TRAMP has an upper limit of 128 pages, which
is a usable size for the majority of cases, including many AI
applications. There is obviously a trade-off here since the
more core that a pointer can address, the less percentage
(though not proportionately less) of that core is available!
There is a second trade-off because the size of the units
which must be addressed determines the number of bits needed
to address them—the larger the unit, the fewer bits required,
but generally, the less efficiently it is used. We arbitrarily
decided that the half-word pointers that TRAMP uses to address
double words are, in a sense, optimal. Should more experience
prove us wrong, or if some special application should require
much greater capacity, the structure could be augmented, e.g.,
to incorporate full 32-bit addresses, with little more trouble
than alteration of an assembly parameter. At this time it is
not anticipated that explicit use will be made of any peri-
pheral storage devices, other than the transparent swapping
performed by the timesharing supervisor.

The sizes of the various Name and Association tables
are another assembly parameter. Currently the 7 tables occupy
4 pages of core. This figure was arrived at arbitrarily and
will remain in force pending feedback which indicates that it
is inappropriate.

TRAMP is initially loaded into core with all of its
tables, a one-page PSECT and 8 pages of GS . Thereafter, when
more space is needed (GS is the only unit that will require
more space, since overflow from the tables is placed in GS),
TRAMP requests it of the system in blocks of 8 pages until
the maximum of 128 is reached, or the system is unable to comply
with the request.

TRAMP is continually generating temporary lists which
are immediately returned to GS when no longer needed. As
well, when an association is destroyed, or a relational defini-
tion erased (KR and KDR , Appendix B), as much storage as is
being released is at that time returned to GS . Thus, un-
used units are never left lying about core: a unit is returned,
not discarded, eliminating formal garbage collections by en-
suring that garbage is never created.

Figure 8. Defined Relation Name Table

H_1 (name)
Name Table
T
H
PB
next
This name table is identical to the other name tables (Figure 6) except for the Pointer Block.
Pointer Block (PB)
Pointer to EBCDIC representation of definition as entered by user
Program for answering F1: #(r1,A,∩,**)
Program for answering F2: #(r1,A,**,V)
List of all other relations that this relation is defined in terms of.
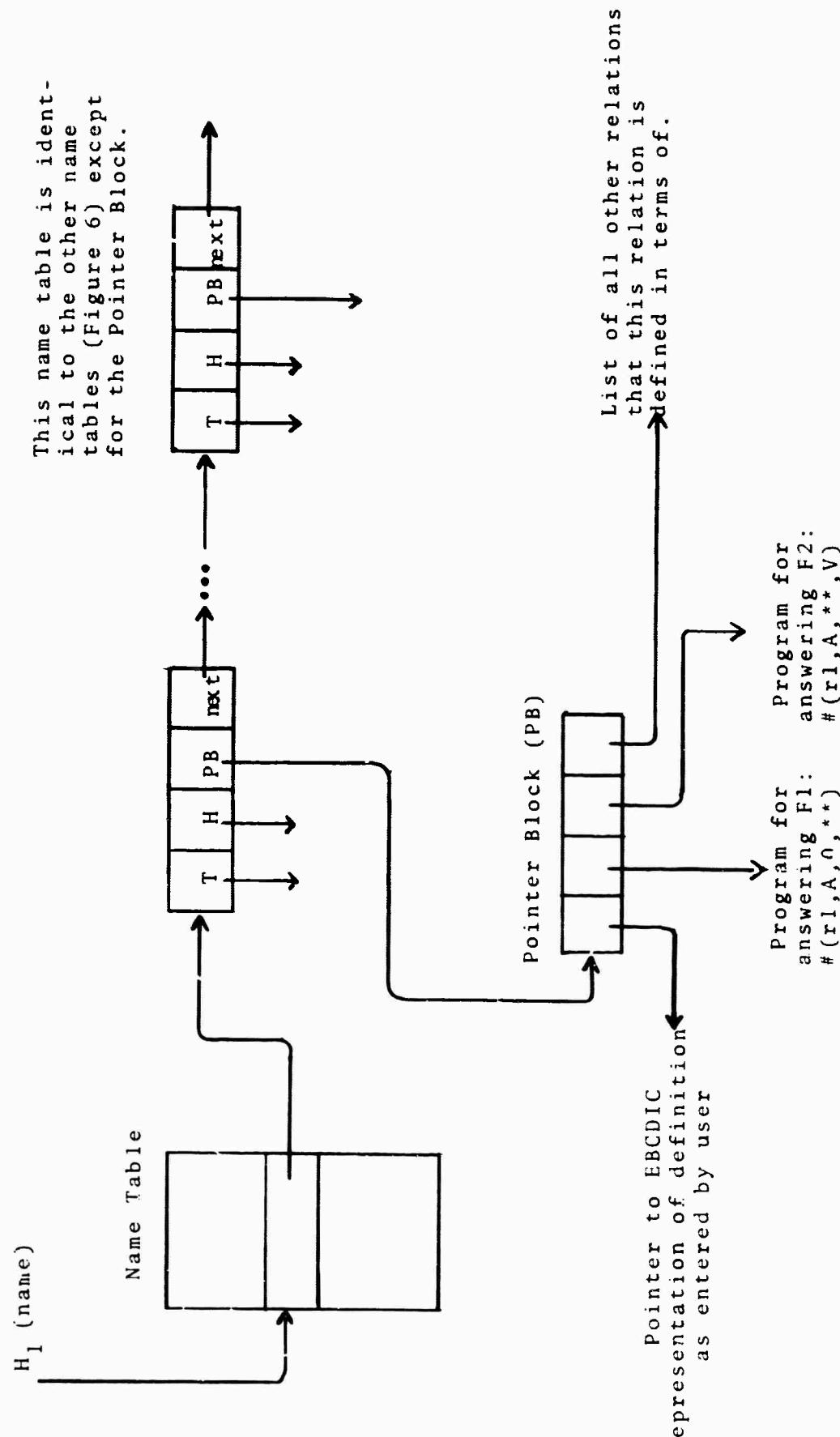
APPENDIX A

UMIST

The following excerpts from the UMIST manual are re-
produced with the permission of Mr. Tad Pinkerton, whose work it
is. What follows is partial and incomplete and is intended
only to familiarize the reader with the structure of the
language and enable him to follow the TRAMP definitions and
examples. A complete description of the UMIST language may
be found in The University of Michigan Terminal System Manual:
MTS, 2nd Ed., Vol II, The University of Michigan Computing
Center, December 1967.

::
::

A level of the TRAC language called "TRAC 64" is
described in [14]. It is the basic standard and point of
reference for UMIST. A good discussion of TRAC 64's design
goals and principles is given in reference [15]. Much of the
motivation for the development of the TRAC language came from
the work of Eastwood and McIroy [2,13] at Bell Laboratories.
A system similar to the TRAC language which was developed in-
dependently in Great Britain is described by Strachey [19].

MODE OF OPERATION

There are two kind of functions: primitives, or
machine-language subroutines that support the system in its
environment. The primitives are the basis for the second type
of function, called forms, or named procedures in UMIST storage,
which are character strings written like macro definitions and
expanded, interpretively, when called. When writing a function
call, one specifies whether its value (replacing the call) is
to be processed again as part of the input string (active call),
or whether processing is to continue starting with the portion
of the string to the right of the value returned (neutral call).

A-1

A single processing cycle is completed when the scanning and
evaluating process reaches the right-hand end of the string.

Sequencing and evaluation in UMIST are inherently
recursive: function calls are evaluated from left to right,
but may be nested to any depth in the arguments of other calls.
Each function call is evaluated when, and only when, all of its
arguments have been completely processed. Thus the string be-
ing processed is divided logically into two parts: the active
string, consisting of input text (possibly preceded by inserted
functional values) which is yet to be scanned, and evaluated
arguments of function calls which are not completely ready for
evaluation. This mode of operation, based on the completely
interpretive execution of function calls, eliminates the dis-
tinction between program and data.

SYNTAX

Each function call in UMIST has the form of a spe-
cially delimited argument list, in which the name of the func-
tion is always the first argument. Calls may be open (a vari-
able number of arguments) or closed. A function call may be
protected from evaluation by the use of literal delimiters.
Another delimiter signals the right-hand end of the input
string. These considerations lead to a syntax in which there
are seven special symbols, whose occurrences are deleted from
the string during syntax scanning and whose presence indicates
the beginning or end of a substring. The character strings
enclosed in brackets below are the UMIST special symbols:

1. Beginning of neutral function call [##(]
2. Beginning of active function call [#(]
3. End of argument [,]
4. End of call [)]
5. Beginning of literal [(]
6. End of literal [)]
7. End of input string [']

Note that the three beginning-of-substring symbols
##(and #(and (are terminated by the occurrence of the same
end-of-substring character,). UMIST has a "parenthesis
balanced" syntax, in the sense that an occurrence of the right
parenthesis matches only the last previous occurrence of any one
of the beginning-of-substring special symbols. Whenever a
literal substring is encountered, the UMIST processor removes
the enclosing parentheses, but only the outer set is removed
if more than one matching pair occurs. Thus a string initially
protected from evaluation may be evaluated if scanned a second
time, and, in general, evaluation can be controlled to occur
the n-th time the substring is scanned.

READ STRING AND PRINT STRING

The value of a 'read string' function call

$$\#(RS)$$

is an input string accepted from the current input device.
The 'print string' function

$$\#(PS,X)$$

causes the display of the second argument, here symbolized by
X , on the current output device, and has a null value.
When the UMIST processor is first given control,
and at the end of every processing cycle, the idling procedure

$$\#\#(PS,\#(RS))$$

is automatically loaded as an input string. This procedure
first causes a read from the input device, with the input string
becoming the second argument of the 'print string' call. Thus
the string, if any, remaining when the input string has been
completely processed, is finally printed before the idling

procedure is again loaded. For example, if the input string is

#(PS,ABC)'

then after the 'read string' has been evaluated the proces-
sor is scanning the string

##(PS,#(PS,ABC))

and the inner call produces the output ABC ; the outer call
nothing, since the inner 'print string' has a null value.

DEFINE, CALL, AND SEGMENT STRING

Any character string in UMIST can be given a name
and placed in storage, from whence it can be called by using
its name. Tne null-valued 'define string' function

#(DS,A,B)

places the string B in storage with the name A . A is
called a <u>form</u> with value B . At most one string can be de-
fined with a given name at any one time: use of the same
name replaces a former definition. The value is retrieved
with the 'call string' function

#(CL,A) .

A form name, like a value, is any character string.
The only restriction on length is that of the total string
capacity of the processor.

The occurrence of strings in storage is deleted
with the 'delete definition' function

#(DD,N1,N2,...) .

This null-valued function removes the names N1,N2,... as
forms and discards their values.

Once defined, a form can be "parameterized," or segmented, using the 'segment string' function:

$$\#(SS,A,X1,X2,\ldots)$$

This null-valued function scans the form A , searching for an occurrence of the string X1 as a substring. If X1 matches a part of A , that part is excluded from further matching, creating a "formal variable," or segment gap. The rest of the form is also compared with X1 to create, if possible, more segment gaps, all of which are assigned the ordinal value one, identifying the argument matched. The (separate) substrings of the form not already taken for segment gaps are next scanned with respect to the string X2 , and any occurrences of the latter substring in A create segment gaps of ordinal value two, etc.

Thus, the 'define string' and 'segment string' functions together create a "macro" in which the segment gaps locate the "formal parameters." The "macro" is expanded by supplying the "actual parameters" in a call on the 'call string' function mentioned above:

$$\#(CL,A,Y1,Y2,\ldots) \ .$$

The value of the 'call' is generated by returning the form A with all the segment gaps of ordinal value 1,2,... replaced by Y1,Y2,... respectively. If extra arguments are given in a CL , they are ignored. If some are missing, null strings are used as their values.

THE EQUAL FUNCTION

A decision function is provided for character strings:

$$\#(EQ,A,B,T,F) \ .$$

If the string A is identical to the string B , then the
value of this function is the fourth argument, T ; otherwise
the value is the fifth argument, F . Since the strings T
and F may be any UMIST procedures, this primitive is the one
normally used for branching.

APPENDIX B

TRAMP FUNCTIONS

BLANK PAGE

APPENDIX B

TRAMP FUNCTIONS

1. This appendix is intended as a reference manual for TRAMP and
    provides full specifications for using the various functions
    available in it. This section assumes a working knowledge
    of UMIST (Appendix A), as well as familiarity with the design
    goals of TRAMP as set forth in the body of this paper.

II. Running TRAMP in MTS (<u>M</u>ichigan <u>T</u>erminal <u>S</u>ystem)

    TRAMP is invoked in the normal way be specifying it as the
    object file of a "RUN" command. The input is taken from the
    logical device SCARDS; output is put on the logical device
    SPRINT; and error comments (TRAMP, not UMIST) appear on the
    logical device SERCOM. While all three are global run para-
    meters, the active input/output devices may be switched from
    SCARDS/SPRINT to some other logical device, dynamically with-
    in UMIST, via the #(par, function.

    The "RUN" command can accept, besides these keyword para-
    meters, a parameter "list" (via "PAR=") consisting of the
    following three global parameters, whose default values
    are underscored:

    a. <u>NOPRIME</u> or PRIME

        This parameter specifies whether or not the prime
        (') will be required to terminate TRAMP input lines.
        If PAR=PRIME , then the program is in the normal UMIST
        mode of operation: an input line is not terminated
        until the prime is encountered. Otherwise, a prime will
        automatically be appended to the end of each input line
        (if not already there), as delineated by a carriage re-
        turn or other device-dependent end-of-record signal, by
        TRAMP before it is passed on to UMIST, which is still

operating in the normal mode. If an input record has
as its last character an ampersand (&) , then that is
taken to be a continuation mark: the ampersand is
deleted from the line, which is passed on to UMIST
without a prime. If the ampersand is followed by a
blank, then it is not a continuation mark; it must be
the last character, not just the last non-blank char-
acter!

The mode of operation is initially set with this
parameter, but may be dynamically altered during exe-
cution via the PRIME function, fully described in this
Appendix.

b. *UMISTL or *UMIST

This parameter specifies which version of UMIST
is to be used as the host interpreter. Presently the
above two files are the two versions of UMIST avail-
able. These two, and any other that might become
available, may be used.

c. NOW or LATER

This parameter specifies when the TRAMP functions
are to be loaded. If PAR=LATER , then only UMIST will
be loaded initially, with the loading of TRAMP being
deferred until a call on #(tramp), further explained
below.

NAME:	DR
PROTOTYPE:	#(DR,A,∅,V)
PURPOSE:	This is the associative storage function—the function that inserts the data into the structure.
DESCRIPTION:	The three arguments, A, ∅ , and V , are each non-empty sets. The set element delimiter in TRAMP is the semicolon (;) because of the important role played by the comma in UMIST. The triple is ordered and interpreted as meaning: A (∅) = V . Each element of each set is grouped with each pair of elements of the other two sets, and the resulting triple is stored, i.e., each point in the cartesian product is stored. The three sets are ordered sets only inasmuch as the order in which they appear in the storage declaration is retained.
	DR simply inserts the data into the structure in a way in which it can be efficiently retrieved. No check is made for inconsistency of data or for redundancies.
EXAMPLES:	#(DR,COLOR,CAR,RED;GREEN)
	#(DR,AGE,MABLE;EUNICE,39)
	this would store: AGE (MABLE) = 39
	AGE (EUNICE) = 39

NAME: KR

PROTOTYPE: #(KR,A,Ø,V)

PURPOSE: To undo what DR did—to erase an association
                from memory.

DESCRIPTION: The syntax of this function is exactly the same,
                and the effect exactly the opposite, of DR .

EXAMPLES: #(KR,COLOR,CAR,CHARTREUSE)

                #(KR,AGE,#(RLR,AGE,**,*X*),#(X))

                This would delete ALL associations containing
                "AGE" as the "A" component [see RLR , below].

NAME: RL

PROTOTYPE: #(RL,A,∅,V)

PURPOSE: This is the associative retrieval function.
                   "Questions" are asked of the data structure
                   by calling RL and specifying which, if any,
                   among A, ∅ , and V are variables.

DESCRIPTION: Variables are denoted by enclosing a name, pos-
                   sibly null, within asteriks (*) . To ask the
                   question: "What color is the car?" , one would
                   write: #(RL,COLOR,CAR,**) or #(RL,COLOR,CAR,
                   NAME) . The "answer set" in this example is
                   the set of all third components of associations
                   having "COLOR" as the first component and
                   "CAR" as the second. In the first instance
                   above, the variable is not Named [nothing
                   between the asterisks]. In this case, the
                   answer set is the Value of the function. In the
                   second instance, the variable is Named, which
                   results in the function being Null Valued, and
                   the answer set being stored in UMIST form
                   storage labeled by the Name within the asterisks.
                   Thus, the following two statements are exactly
                   equivalent:

                        #(DS,ANS,#(RL,COLOR,CAR,**))

                        #(RL,COLOR,CAR,*ANS*)

                   If there are no variables, e.g., #(RL,COLOR,
                   CAR,RED) , then the question being asked is:
                   "Does A(∅) = V?", or in this case, "Is the
                   car colored red?" No answer set is generated,
                   rather a "truth value" is returned as the
                   value of the function. If the specified associa-
                   tion is in f ct resident in the structure, or

derivable thereof, then the value is "1"; if
not, the value is "0." An ambiguity arises
when one or more of the three sets has cardi-
nality greater than one. Suppose #(DR,COLOR,
CAR,RED) had been entered. Then,

#(RL,COLOR,CAR,RED)   would have the value "1"

#(RL,COLOR,CAR,BLUE)   "   "   "   "   "0"

#(RL,COLOR,CAR,RED;BLUE)"   "   "   "   "?"

That is, the first association is found in storage,
and the answer is "1." The second is not found,
and the answer is "0." But two associations are
specified by the last example, one is verified,
the other not, and TRAMP returns the value "?"

If there is one variable, then TRAMP is being
asked to "fill in the blank." The one variable
may be in any of the three positions of the
triple. The variable may be either Named or
Unnamed, with the respective consequences des-
cribed above.

If there are two variables, then two answer sets
are generated. One of the variables is picked as
the index variable, and values are one-by-one
substituted for it, internally iterating on the
one-variable question. The one constant may
again be in any of the three positions of the
triple. If both variables are named, the func-
tion is Null Valued, and the two answer sets
are stored and labeled by their respective
names. If one is Named and the other Unnamed,
then the set corresponding to the Named variable
is stored and the other answer set is the Value

of the function. It is syntactically valid for
both variables to be Unnamed, but this should
not be done since then the Value of the function
would be the concatenation, not union, of the
two answer sets.

The two-variable questions generate two answer
sets—not a set of ordered pairs! Soon a varia-
tion of this function may be offered which will
allow the generation of ordered pairs. In the
meantime, if this is desired, the user will have
to write a short UMIST procedure to pick out
the proper subset of the cartesian product of the
two answer sets.

The present form of the two-variable questions—
generating two answer sets—is very often used
to find all "objects" associated with some other
"objects," without regard for the third component
of the triple. For example, to find all those who
have sons, one could say:

$$\#(r1,SON,**,*X*)$$

with the set of all sons now being stored in the
form "X." In general, this generated set, here
the set X , will not be further used and is not
wanted. TRAMP recognizes one special Named
Variable for two-variable questions: "@," as
denoting that the corresponding answer set is
not to be generated. Thus,

#(r1,SON,**,*SONS*)   would return the set of all
                       those who have sons, and
                       store the set of all Sons
                       in the form "SONS"

B-8

#(rl,SON,**,*@*)   would likewise return the
                     set of all those who have
                     sons, but would <u>discard</u> the
                     set of Sons.

If there are three variables it is interpreted
as being a request for a dump of the associative
memory. If any of the three variables are Named,
the names are ignored. Alternatively, one can
simply call: #(DUMP).

EXAMPLES: #(RL,*REL*,JOHN,HARVEY)   put the set of all re-
                                          lations that associate
                                          John with Harvey in the
                                          string "REL."

              #(RL,SON,CLYDE,**)   return the set of
                                          Clyde's sons.

              #(RL,COLOR,**,*COLOR*)   return the set of all
                                          objects that have the
                                          attribute "COLOR," and
                                          place the set of all
                                          colors in the string
                                          "COLOR."

              #(RL,*X*,*Y*,*Z*)   give a dump of the as-
                                          sociative memory. The
                                          three names are ignored.

              #(RL,AGE,#(RL,FA,#(RL,WIFE,#(RL,BRO,MARY,**),
               WIVES)#(WIVES),**),**)   recursively asks the
                                          question: "How old are
                                          the fathers of the wives
                                          of Mary's brothers?"
                                          Also, the set of the
                                          wives of Mary's brothers
                                          is now in the string,
                                          "WIVES."

              #(rl,COLOR,**,*@*)   return the set of all ob-
                                          jects that have the at-
                                          tribute "COLOR," but <u>do
                                          not</u> generate the set of
                                          colors.

#(r1,COLOR,CAR,*@*)   put the set of the colors
of the car in the string
"@." "@" is a special
symbol only in the two-
variable questions.

NAME: RLR

PROTOTYPE: #(RLR,A,∅,V)

PURPOSE: To retrieve answer sets that may contain re-
                dundancies.

DESCRIPTION: This function is identical to RL   except that
                any redundancies are reported. RL   returns non-
                redundant answer sets, while RLR does not check
                for redundancies, and is therefore significantly
                faster.

EXAMPLES: #(RLR,AGE,EUNICE;GLADYS,**)

                         if they are both 31, then the value will
                         be: "31;31."

NAME: INT

PROTOTYPE: #(INT,A,O,V)

PURPOSE: To generate intersections of answer sets.

DESCRIPTION: This function has the same syntax as the one-
                 variable question of RL . RL generates the
                 union of the answer sets, while INT generates
                 the intersection

                 #(r1,SOUTH;WEST,TOLEDO,**)   generates the set of
                                              all things either
                                              south or west of
                                              Toledo.

                 #(int,SOUTH;WEST,TOLEDO**)   generates the set
                                              of all things both
                                              south and west of
                                              Toledo.

                 If both constant sets are singletons, INT and
                 RL will yield identical answer sets. The vari-
                 able may again be in any of the three positions
                 and may be either Named or Unnamed. This func-
                 tion must have exactly one variable.

EXAMPLES: #(int,NORTH;EAST,CHICAGO,*NE*)

                         place the set of everything both north
                         and east of Chicago in the string "NE."

                 #(int,**,JOHN;MARY,CLARA)

                         return the set of all relations that
                         John and Mary commonly share with Clara.

NAME: RCOM

PROTOTYPE: #(RCOM,SET1,SET2,NAME)

PURPOSE: To compute the relative complement of two TRAMP
                 sets.

DESCRIPTION: The third argument is logically subtracted
                 from the second argument, with the disposition
                 of the resulting set determined by the fourth
                 argument: if it is present, the function is null
                 valued and the set is stored in UMIST form
                 storage labeled by the name; if the fourth
                 argument is omitted, the relative complement
                 of the other two arguments is returned as the
                 value of the function. The set computed con-
                 sists of all elements of "SET1" that are not
                 elements of "SET2."

EXAMPLE: #(RCOM,#(RL,AGE,**,40),#(RL,SPOUSE,**,*@*),
                 SPINSTER)

                         this would store in the string
                         "SPINSTER" all those who are 40
                         years old and not married.

B-13

NAME: SYMD

PROTOTYPE: #(SYMD,SET1,SET2,NAME)

PURPOSE: To compute the symmetric difference of two sets.

DESCRIPTION: The symmetric difference of two sets is defined
                to be the set of all things that are in either
                of the two sets, but not in both (exclusive OR).
                The syntax of SYMD is identical to that of RCOM,
                with the fourth argument determining what will
                be done with the generated set.

EXAMPLE: #(SYMD,#(RL,BRO,**,*@*),#(RL,SIS,**,*@*))

                this would return the set of all those who
                have siblings, but siblings of only one
                sex.

NAME:	INT
PROTOTYPE:	#(INT,SET1,SET2,NAME)
PURPOSE:	To intersect two TRAMP sets.
DESCRIPTION:	This function has the same syntax as the other two set operators: RCOM and SYMD: the two operands are the second and third arguments (SET1 and SET2) and the fourth argument specifies the disposition of the result: if it is present, it will be used as the name of the form into which the answer will be placed; if omitted, the answer will be returned as the value of the function. The answer is a straight set inter-section, except that any redundancies (e.g., introduced by a call on RLR) are deleted.

Note that this function has the same name as the retrieval function INT. There is no ambiguity and there should be no confusion, since the two func-tions have dissimilar syntax! The retrieval function INT is called by specifying exactly three functional arguments, of which exactly one is a Variable; the Set Operation INT is invoked by giving either two or three functional argu-ments, of which exactly zero are Variables! I.e., a variable specifies retrieval—if there is no variable then a question is not being asked.

EXAMPLES:

                    #(int,#(int,NORTH;EAST,CHICAGO,**),

                    #(int,SOUTH;WEST,MAINE,**),UNHUH)
recursively uses both forms of INT to place the
set of all things both northeast of Chicago and
southwest of Maine in the string "UNHUH."

#(int,#(X),#(X),X)   remove all redundancies from
                     the string "X."

#(int,#(rl,AUTHOR,**,GEORGE),#(rl,SUBJECT,**,SEA))

        return the set of everything George wrote
        about the sea.

NAME: DUMP

PROTOTYPE: #(DUMP)   or   #(RL,**,**,**)

PURPOSE: To obtain a complete listing of everything
                that is explicitly stored in the data structure.

DESCRIPTION: All associations explicitly stored are printed
                out, using the "A (∅) = V" format. A and ∅
                are singletons and V is the set of all "values"
                associated with the A/∅ pair. Any redundancies
                in the V set are printed. Implied associations
                are not listed in the dump. After all of the as-
                sociations are listed, all of the current rela-
                tional definitions [entered by D.R , below]
                are displayed.

EXAMPLES: #(DUMP)

                #(RL,**,** **)

                #(RLR,*X*,**,**)

NAME: ERM

PROTOTYPE: #(ERM)

PURPOSE: To completely erase the memory for a fresh
                restart.

DESCRIPTION: It is not anticipated that this fucntion will
                he called very often, if ever, and to prevent
                its being invoked unintentionally, via mis-
                spelling, etc., confirmation is required by
                TRAMP before it actually erases the structure.
                This is similar in form and in content to the
                confirmation that MTS requires before EMPTYing
                a file: an exclamation point (!) , or the
                two letters "ØK" are positive confirmation.
                Anything else cancels the request.

NAME: USE

PROTOTYPE: #(USE,NAME)

PURPOSE: To obtain the number of explicit associations that the NAME is used in.

DESCRIPTION: The value of the function is the total number of associations that the name in the argument is used in. Any implied associations are not included in the USE count. There is no breakdown as to how the name is used within the associations, simply a count of the triples in which it appears.

EXAMPLES: #(USE,JOHN)

              #(USE,COLOR)

                 How many objects have the attribute color?

NAME:	CT
PROTOTYPE:	#(CT,SET)
PURPOSE:	To determine the cardinality of a TRAMF set.
DESCRIPTION:	This is a very simple function that is significantly faster and more convenient than a UMIST procedure that would do the same thing. It distinguishes between: a missing argument; a null set; and a singleton (set with no semicolons); but otherwise is simply an efficient way to count semicolons.
EXAMPLES:	#(CT,#(SET)) what is the cardinality of "SET?"
	#(CT,#(RL,SON;DAUGHTER,SHERMAN,**))
	How many children does Sherman have?

NAME:	TABLE
PROTOTYPE:	#(TABLE,X)
PURPOSE:	To obtain the contents of one of the Name Tables.
DESCRIPTION:	The argument, X , specifies which of the four Name Tables is desired:

    A   -   attribute
    Ø   -   object
    V   -   value
    D   -   defined relation

The set of names found on the particular table is returned as the value of the function.

EXAMPLES: #(TABLE,A)

#(TABLE,D)

#(rcom,#(table,A),#(table,D))

> return the set of all names that have been used as "attributes" but have not been given definitions. The Defined Relation Name Table is always a subset of the "A Name Table!"

#(int,#(int,#(table,A),#(table,O)),#(table,V))

> return the set of all things that have been used at some time in each of the three positions of the associative triple.

#(symd,#(table,A),#(table,V))

> return the set of all names used as either "attributes" or "values" but not as both.

NAME: PRIME

PROTOTYPE: #(PRIME,[ON,OFF])

PURPOSE: To set or invert the mode of operation regard-
                ing the prime that terminates all UMIST input
                lines.

DESCRIPTION: An internal switch in TRAMP determines whether
                or not a prime is required to terminate an in-
                put line. This switch is initially set by the
                parameter in the RUN command. The function
                PRIME may be used to alter dynamically the set-
                ting of this switch during execution.

                The argument to PRIME may specify that this
                switch is to be turned ON or OFF, or simply
                inverted from its present setting. #(PRIME,ON)
                turns the prime ON, i.e., it specifies that a
                prime will be required to mark the end of a
                line. #(PRIME,OFF) sets the switch the other
                way; equivalent to: PAR=NOPRIME. Full details
                of operation with the switch off appear in the
                introduction to this Appendix.

EXAMPLES: #(PRIME,OFF)

                #(PRIME)

                #(PRIME,ON)

                #(PRIME,X)   an unrecognizable argument inverts
                             the switch.

NAME: TRAMP

PROTOTYPE: #(TRAMP)

PURPOSE: To load TRAMP if PAR specified that loading was
                to be delayed.

DESCRIPTION: If PAR=LATER, then only UMIST will be loaded
                initially. When ready for TRAMP, the user
                issues this function call which loads and links
                up all the TRAMP functions.

                The function "TRAMP" is defined only when PAR=
                LATER, and then only until it has been called.

NAME: SAVE

PROTOTYPE: #(SAVE,FDNAME,RLNG,ID)

PURPOSE: To save the current state of the data structure on an auxiliary device so that at a later date the structure can be initialized to contain the present data.

DESCRIPTION: FDNAME is the name of the file or device onto which the data are to be SAVEd. RLNG is an optional argument specifying the record lengths to be written. If this argument is either omitted or specifies too large a record length for the particular device, the following default values, which are the respective physical maximums, will be used:

PUNCH	80
FILE	255
TAPE	32,760

If RLNG = 80, either explicitly or by default, then each record will contain 72 bytes of information and 8 characters of sequential identification, the first 4 of which may optionally be specified in the last argument, ID. If more than 4 characters are given, extra characters on the right will be truncated. If less than 4, trailing blanks will be appended. If RLNG=80 and this argument is omitted, the 4-character MTS signon ID will be used. If RLNG = λ , $\lambda \neq 80$, then there will be λ bytes of information with no identification.

EXAMPLES: #(SAVE,MYFILE)

                        write 255-byte records into the file

                 #(SAVE,MYFILE,80,IDX)

                        write 80-byte records into the file with
                        the specified ID. Can now be copied to
                        a card punch.

                 #(SAVE,*PUNCH*,,IDZ)

                        punch the data onto cards with "IDZ" ID.

                 #(SAVE,*PDN1*,80)

                        write 80-byte records onto tape using
                        MTS ID.

                 #(SAVE,*PDN2*,255)

                        write records on tape that can be copied
                        into a file.

NAME: COPY

PROTOTYPE: #(COPY,FDNAME)

PURPOSE: To read back in what has previously been SAVEd.

DESCRIPTION: COPYing in a new structure completely erases
                anything that might be in the structure at the
                time the COPY is called. There is no direct way
                to merge two TRAMP data files. (See Appendix C.)

NAME: PAGE

PROTOTYPE: #(PAGE)

PURPOSE: To ascertain what size file will be required
                to SAVE in and/or how much core the data
                structure is occupying.

DESCRIPTION: PAGE is a null-valued printing function which
                prints on the current output device. The out-
                put is the number of pages currently in core
                that will have to be saved, and how many exten-
                sions have been made to TRAMP. The sizes of
                the various tables used by TRAMP are assembly
                parameters and are likely to change. Presently
                the tables occupy a total of 4 pages of core.
                The information printed by PAGE is the amount of
                core being used in addition to the tables (tables
                cannot grow during execution).

                TRAMP is initially loaded with an Available Stor-
                age List 8 pages long (32, 768 bytes). As this
                is used up, more is acquired from the system in
                blocks of 8 pages, called extensions. There can
                be up to 16 extensions (presently meaning that
                a maximum of 132 pages = 540,672 bytes would
                have to be SAVEd). These 8-page blocks are
                never broken up—SAVEing requires that the
                entire block(s) be written. In summary, (assum-
                ming 4 pages for tables) there is a minimum of
                12 pages (= 49,152 bytes) and a maximum of 132
                pages (= 540,672 bytes), with the minimum ap-
                proaching the maximum in steps of 32,768 bytes.

NAME: DDR

PROTOTYPE: #(DDR,(REL = EXP))   or   #(DDR,(REL := EXP))

PURPOSE: To define a relation in terms of other rela-
                  tions, thereby creating implicit associations
                  in the data structure.

DESCRIPTION: REL is the relation being defined, and EXP
                  is an expression which is the definition. The
                  equal sign is the delimiter and must be present.
                  In the prototype the entire argument to DDR is
                  enclosed in parentheses, i.e., a UMIST "literal."
                  Depending on the particular definition, this
                  may or may not be necessary, but it will never
                  hurt, and it is good practice always to paren-
                  thesize the argument.

                  EXP is composed of one or more relations joined
                  by the logical connectives: .A. conjunction;
                  .V. disjunction; .N. negation; two relational
                  operators: / (slash) relative product or com-
                  position; .CON. converse; and six equality
                  operators: .EQ.; .NE.; .GE.; .LE.; .GT.; .LT.; with
                  obvious meanings.

                  The "R(x,y)" format is the relational format
                  adopted by TRAMP and is interpreted to mean
                  that R (x) = y .

                  "Converse" simply inverts the order of the two
                  relational arguments: $R(x,y) \leftrightarrow .CON.R(y,x)$.
                  Thus, "child of" is the converse of "parent of,"
                  any symmetric relation is its own converse, etc.

                  Composition is defined: $\forall x \, \forall y \, \exists z [(S/T)(x,y) \leftrightarrow$
                  $S(x,z) \wedge T(z,y)]$. Specifically, the declaration:
                  #(DDR,(R:= S/T)) would tell TRAMP that R(x,y)
                  if for some z: S(x,z) and T(z,y).

DDR is the only TRAMP function that allows spurious blanks.
Before compiling the definition, all blanks are removed. In
all other functions (except EDIT, below, which is another
entry to this function), blanks are valid EBCDIC characters
and are treated like any other. Definitions may be either
abbreviated: #(DDR,(R1 = R2 .V. R3)) ; or in an expanded form:
#(DDR,(R1(X,Y) = R2(X,Y) .V. R3(X,Y))). There is the restric-
tion that any one definition be consistent. #(DDR,(R1 = R2(X,Y)))
is not legal. The two relational operators, composition and
converse, may be used only in abbreviated definitions where there
are no explicit relational arguments. On the other hand, the
equality operators may only be used with relational arguments
as their operands. A constant which is to be use. as a rela-
tional argument is denoted by enclosing the name of the constant
in double quotes (").

Precedence of operators: The precedence ordering of
the various operators is as follows:

/	composition
.CON.	converse
.EQ. etc.	all equality operators have the same precedence
.N.	negation
.A.	conjunction
.V.	disjunction

The above precedence ordering may be altered in the usual way
by the appropriate use of parentheses.

One major constraint is placed on the argument to
DDR: relations must be defined so that at least one set is
generated. The intent of this constraint is that there be
at least one reference set. The "whole space" may never
be the reference set!

    #(DDR,(R = .N. S))   is illegal, since it specifies a
                         global complement, i.e., it references
                         the "whole space."

```
#(DDR,(R = .N. S .A. T))     is legal, since it first generates
                             a reference set via  T , and then
                             places a constraint on that reference
                             set, not on the whole space.
```

Besides the choice of using the "expanded" vs
"abbreviated" notation for defining a relation, the user has
the option of specifying whether or not the implication is one
way, or specifies an if and only if condition.

Husband = .CON. Wife

is an iff condition whereby it is implicit that:

Wife = .CON. Husband

the equal sign will be used to denote this kind of equivalence
which can be interpreted as meaning iff.

On the other hand,

Parent := Father or Mother

is one-way implication, giving information about the relation
"parent" while giving none about the relations "father" or
"mother." To denote this the "assignment symbol" (:=) is used.

In general, if the assignment symbol is used then no
attempt will be made to extract information about the relations
on the right side of the equation. If the equality symbol is
used, such an attempt will be made. This attempt will not,
of course, always be successful:

Parent = Father .V. Mother

gives us no information about "father" or"mother" even through
the equality sign is used.

If a relation is defined, and later a new definition
is given for that same relation, TRAMP simply OR's the two
definitions together. If there is a syntactic error in the
second definition, a diagnostic is printed and the earlier
definition is retained.

EXAMPLES:

#(DDR,(BIGGER = BIGGER / BIGGER))

    BIGGER is transitive

#(DDR,(BIGGER(A,B) = BIGGER(A,C) .A. BIGGER(C,B)))

    same definition using expanded format.*

#(DDR,(SIB = BRO .V. SIS .V. .CON.SIB))

    a sibling is defined to be a brother or a sister
and it is symmetric

#(DDR,(HUSBAND = .CON. WIFE))

    husband is the converse of wife

#(DDR,(BRO(CAIN,ABEL) = SIB(CAIN,ABEL) .A. SEX(ABEL,"MALE")))

    a brother is a male sibling. Constants are enclosed
in double quotes

#(DDR,(BIGGER = LARGER))

    bigger and larger are synonymous

#(DDR,(MALE(X) = SEX(X,"MALE")))

    unary relations may be defined

#(DDR,(BROTHER(X,Y) = FATHER(X,Z) .A. FATHER(Y,Z) .A. MALE(Y)
               .A.X.NE.Y))

    a brother can be defined as a male offspring of the
same father, other than oneself

#(DDR,(PARENT = FATHER .V. MOTHER))

#(DDR,(NEPHEW = SIBLING / SON))

    a nephew is the composition of sibling and son.

#(DDR,(UNCLE = .CON.(SIBLING / SON)))

    in a male world, uncle is the converse of nephew and
may be defined as the converse of the definition of
nephew.

#(DDR,(UNCLE = .CON. NEPHEW))

    or simply as the converse of nephew.

---

* The dummy arguments used in expanded definitions may be
up to eight characters long.

#(DDR,(STEPMOTHER = FATHER / SPOUSE .A. .N.MOTHER))

a stepmother is the spouse of the father who is not
the mother.

NAME:	KDR
PROTOTYPE:	#(KDR,REL1,REL2, ...,RELn)
PURPOSE:	To erase definitions made by DDR .
DESCRIPTION:	KDR may have any number of arguments. The definition for each of the relation names given as arguments is deleted.
EXAMPLES:	#(KDR,SIBLING)
	#(KDR,NEPHEW,UNCLE,SPOUSE)
	#(DS,X,#(TABLE,D))#(SS,X,;)#(KDR,#(X,(,)))
	would erase ALL definitions

NAME: SHOW

PROTOTYPE: #(SHOW,RELATION)

PURPOSE: To display the current definition of a relation.

DESCRIPTION: SHOW will display the definition of the relation specified by its argument exactly as it was entered by the user, except that blanks will have been removed. If more than one definition has been given for the r lation, they will all be concatenated, separated by a break character, and displayed in a continuous line. Three actions may be taken by SHOW : if the relation has been successfully defined, its definition will be displayed on the current output device; if TRAMP has never heard of the relation, the comment: "RELATION XXX HAS NOT BEEN DEFINED." will be printed; if the relation was unsuccessfully defined, or was erased, via KDR, the comment: "RELATION XXX IS UNDEFINED." will be printed.

NAME: EDIT

PROTOTYPE: #(EDIT,RELATION,PATTERN,REPLACEMENT)

PURPOSE: To correct or alter a relational definition made by DDR .

DESCRIPTION: The second argument is the name of the relation that is to be EDITed. The third argument is the pattern within the definition, as displayed by SHOW, that is to be altered. If this argument is null, it matches the void immediately to the right of the relation name in the definition string. The last argument is the string that replaces the pattern specified by the third argument. Any blanks in the PATTERN or REPLACE-MENT will be ignored. If the last argument is omitted, the pattern is simply deleted. If the string specified by the third argument occurs more than once in the definition, only the first occurrence is changed.

Calling EDIT implicitly calls SHOW to display the EDITed definition.

EXAMPLES: #(EDIT,SIB,(.V.),(.A.))

change the first OR to AND in the definition of SIB . Like DDR , it is good practice to enclose the argument in parentheses.

#(EDIT,REL,(.A. R4))

delete the string ".A.R4" from the definition of REL.

NAME: DDEF

PROTOTYPE: #(DDEF)

PURPOSE: To display all current relational definitions.

DESCRIPTION: DDEF iteratively calls on SHOW for each name
found in the Name Table of Defined Relations.
DDEF is an entry to the second half of DUMP, which
bypasses the listing of the associations.

TRAMP in no way alters anything internal to UMIST.
There are three UMIST functions that cannot be called within
TRAMP since they would render it unusable:

    #(DSS)   Define Special Symbol
    #(DA)   Delete All (including TRAMP)
    #(RES)   Restart - contains #(DA) as a subset.

With these three exceptions, the full facilities of UMIST
are available to the user.

APPENDIX C

EXAMPLES OF TRAMP

APPENDIX C

EXAMPLES OF TRAMP

This Appendix gives both simple and more complicated examples of the use of TRAMP. The most relevant information about UMIST necessary to understanding the examples is found in Appendix A. Complete understanding of the more complicated example of the question-answering program (Fig. 11b) requires more detailed knowledge of UMIST, which may be found in the cited references.

The first set of examples (Fig. 9) show the "FO" questions of Part III. First, we define the COLOR of CAR to be RED and GREEN. Then the various Boolean questions are asked; the answers given by the system are on the line immediately following the "question."

The next examples (Fig. 10a) describe the family tree information of Fig. 1. Two relatively complicated questions are then asked. The first may be stated as: "Who are the people who have brothers whose age is 64?" To illustrate this in depth, we first ask "Who is aged 64?", then "Who are the people who can call this 64-year-old 'brother'?" Obviously, for one of these people (Mary), there is no answer, for no one calls her "brother." The second question asks: "What is the age of the father of the brother of Melissa's mother?" Store the answer (this age) in the form called "NUM." Now Melissa's mother is Edith, Edith's brother is Arnold, Arnold's father is John, who is 64 years old. Hence, a call for the string NUM prints "64."

C-1

```
#(DR, COLOR, CAR, RED; GREEN)'
#(RL, COLOR, CAR, BLUE)'
0
#(RL, COLOR, CAR, RED; GREEN)'
1
#(RL, COLOR, CAR, GREEN; RED)'
1
#(RL, COLOR, CAR, RED; BLUE)'
?
#(RL, COLOR, CAR, RED)'
1
#(RL, COLOR, CAR, RED; BLUE; GREEN)'
?
#(RL, COLOR, CAR, YELLOW; BLUE)'
0
```

Figure 9.   "F0"   Questions.

```
#(DR,FATHER,ARNOLD,JOHN)
#(DR,FATHER,JAMES,ARNOLD)
#(DR,BROTHER,SAM;JOAN,JOHN)
#(DR,BROTHER,EDITH,ARNOLD)
#(DR,MOTHER,ARNOLD,MARY)
#(DR,MOTHER,MELISSA,EDITH)
#(DR,AGE,JOHN;MARY,64)
#(DR,AGE,ARNOLD,39)
#(DR,AGE,EDITH,33)'
#(RL,BROTHER,**,#(RL,AGE,**,64))'
SAM;JOAN
#(RL,AGE,**,64)'
JOHN;MARY
#(RL,BROTHER,**,JOHN)'
SAM;JOAN
#(RL,BROTHER,**,MARY)'

#(RL,AGE,#(RL,FATHER,#(RL,BROTHER,#(RL,MOTHER,MELISSA,**),**),
**),*NUM*)'
#(CL,NUM)'
64
```

Figure 10a.   Family Relationships / Nested Questions.

```
#(DR,SISTER,JOAN,ALICE)'
#(DDR,(SIB = BROTHER .V. SISTER .V. .CON.SIB))'
#(RL,SIB,ALICE,**)'
JOAN
#(DDR,(SIB(X,Y) = SIB(X,Z) .A. SIB(Y,Z) .A. X.NE.Y))'
#(RL,SIB,ALICE,**)'
JOAN;JOHN;SAM
```

Figure 10b.   Relational Example, Associations Between Siblings.

The examples of Fig. 10b show the first use of <u>ddr</u>, where we add to the family of Fig. 10a the sisterhood of Joan and Alice. The first definition of Sibling now allows the retrieval of one of Alice's siblings, though Sibling has never appeared explicitly in an association. The second definition entered completes the job, and TRAMP is now able to return all of Alice's siblings.

As a more complicated example of the use of TRAMP, a rudimentary "question-answering system," with thesaurus, was coded. It should be noted that this illustration is not intended to constitute a good system—in fact, it represents a total of less than one hour's coding and debugging time.

The job of the system is to parse input commands and, from them, generate TRAMP statements. In this regard the system is grossly incomplete, i.e., it uses a most unsophisticated parsing algorithm. The generated TRAMP calls are realistic, nonetheless. The complete program, used as the question-answering system (QAS), is shown in Fig. 11b.* The output from an actual session is shown in Fig. 11a.

To enable the reader easily to follow the dialogue, each statement issued by the QAS is initiated by the word: "ANSWER." Everything else was typed at the terminal by the user. QAS has been given a thesaurus to relax the format of

---

* The code for the question-answering program is shown as it would appear for a processor for the standard TRAC T-64 language, rather than for the locally used dialect UMIST, but the differences are very slight indeed.

C-5

```
INPUT AUTHOR OF HUCKFINN WAS MARKTWAIN'
SYNO.YM MARKTWAIN IS SAMCLEMENS'
WHAT DID SAMCLEMENS WRITE ?'
 ANSWER:
SYNONYM WRITE = AUTHOR'
WHAT DID SAMCLEMENS WRITE ?.
 ANSWER: HUCKFINN
DID TWAIN WRITE HUCKFINN ?'
 ANSWER: NO
SYNONYM MARKTWAIN AND TWAIN'
DID TWAIN WRITE HUCKFINN ?'
 ANSWER: YES
INPUT AUTHOR OF TOMSAWYER IS TWAIN'
INPUT AUTHOR OF THESTRANGER WAS SAMCLEMENS'
HOWMANY BOOKS DID MARKTWAIN WRITE ?'
 ANSWER: 3
WHAT DID TWAIN WRITE ?'
 ANSWER: HUCKFINN; TOMSAWYER; THESTRANGER
INPUT MODEL OF /360 IS 67'
SYNONYM /360 IS IBM'
WHICH MODEL OF IBM DO WE HAVE ?'
 ANSWER: 67
```

Figure 11a.   Output of Question-Answering Program.

QUESTION-ANSWERING PROGRAM

```
#(DS,START,(#(DS,FF,#(RS))#(SS,FF, )#(CL,##(CS,FF))#(CL,START)))'

#(DS,SYN,(#(DS,0,X)#(EQ,##(CC,0),*,X,(##(RL,SYN,X,**)))))#(SS,SYN,X)'

#(DS,CK,(##(RL,SYN,X,*0*)#(EQ,##(CL,0),,(#(DR,SYN,X,X)X),
    (##(CL,0)))))#(SS,CK,X)'

#(DS,DR,(#(DR,#(CL,CK,A),#(CL,CK,O),#(CL,CK,V)))#(SS,DR,A,O,V)'

#(DS,RL,(#(RL,#(CL,SYN,A),#(CL,SYN,O),#(CL,SYN,V)))#(SS,RL,A,O,V)'

#(DS,INPUT,(#(CL,DR,##(CS,FF),#(PS,,##(CS,FF))##(CS,FF),
    #(PS,,##(CS,FF))##(CS,FF))))'

#(DS,SYNONYM,(#(DS,0,##(CS,FF))#(PS,,##(CS,FF))#(DR,SYN,##(CS,FF)
    ,##(CL,0))#(DR,SYN,##(CL,0),##(CL,0))))'

#(DS,WHAT,(#(PS, ANSWER: #(PS,,##(CS,FF))#(DS,0,##(CS,FF))
    #(CL,RL,##(CS,FF),**,##(CL,0)))))'

#(DS,WHO,(#(PS, ANSWER: #(PS,,##(CS,FF))#(CL,RL,##(CS,FF),
    #(PS,,##(CS,FF))##(CS,FF),**))))'

#(DS,HOWMANY,(#(PS, ANSWER: #(PS,,##(CS,FF)##(CS,FF))
    #(CT,#(CL,RL,#(DS,0,##(CS,FF))##(CS,FF),**,##(CL,0))))))'

#(DS,IS,(#(PS, ANSWER: #(EQ,#(CL,RL,##(CS,FF),#(PS,,##(CS,FF))
    ##(CS,FF),##(CS,FF)),0,NO,YES))))'

#(DS,DID,(#(PS, ANSWER: #(DS,0,##(CS,FF))#(EQ,#(CL,RL,##(CS,FF),
    ##(CS,FF),##(CL,0)),0,NO,YES))))'

#(DS,WHICH,(#(PS, ANSWER: #(CL,RL,##(CS,FF),#(PS,,##(CS,FF))
    ##(CS,FF),**))))'

#(CL,START)'
```

Figure 11b.   Question-Answering Program.

statements. This thesaurus, as well as all data, is held in the TRAMP structure. To make a thesaurus entry, the user types the command "SYNONYM" followed by the two synonymous names. A datum is entered by the command "INPUT." The questions are self-explanatory.

Merging TRAMP Data Files

The following procedure shows one way that two TRAMP data files can be merged. This sort of thing is necessary because the COPY function erases the current memory while it is writing in a new one. Thus, assume that files DATA1 and DATA2 are two TRAMP data files that are to be merged:

```
$RUN TRAMP PAR=NOPRIME
#(copy,DATA1)
#(par,FDO,SCRATCH)#(dump)#(par,FDO,*SINK*)
#(copy,DATA2)
#(ds,PARSE,(#(ds,X,##(rs))#(eq,##(cc,X), ,(#(ss,X, )&
#(dr,##(cs,X),#(cs,X),#(nl,##(cs,X))##(cs,X))#(PARSE)),&
(#(par,FDI,*SOURCE*)))))
#(par,FDI,SCRATCH((2)))#(PARSE)
```

By way of a superficial explanation: the procedure "PARSE" simply reads in the "dumped" information and parses those lines to extract the three arguments to DR. The process must start at line #2 because the first line of a dump is always a label. The procedure recursively calls on itself until another "label" is encountered, signaled by not having a blank

in the first column. Segmenting the dumped line for blanks,
the first segment will be the "A" component; the second seg-
ment will be the "O" component (stripped of the parentheses
by calling CS actively); the next segment will be the equal
sign and is discarded via the NL (null) function; the last
segment is the "V" component. Note that PAR must equal NOPRIME
(default case) in order for PARSE to be able to read the dumped
lines in sequence, since they do not have primes on the end.

An analogous procedure could be written and called
by PARSE to read in the rest of the dump, which would contain
the relational definitions, and make those definitions via
generated calls on DDR.

REFERENCES

1. Dodd, G.G., "APL—A Language for Associative Data Handling in PL/I," _Proc. AFIPS FJCC_, November 1966, pp. 677-684.

2. Eastwood, D.E., and M.D. McIlroy, "Macro Compiler Modification of SAP," Computer Laboratory Memo, Bell Telephone Laboratories, Murray Hill, N.J., September 1959.

3. Feldman, J.A., "Aspects of Associative Processing," Lincoln Laboratories, Lexington, Mass., April 1965.

4. Feldman, J.A., and P.D. Rovner, "An Associative Processing System for Conventional Digital Computers," Lincoln Laboratories, Lexington, Mass., April 1967.

5. Gray, J.C., "Compound Data Structure for Computer Aided Design: A Survey," _ACM National Conference_, August 1967, pp. 355-365.

6. Green, B.F., A.K. Wolf, C. Chomsky, K. Laughery, "BASEBALL: An Automatic Question Answer," _Computers and Thought_, ed. E.A. Feigenbaum and J. Feldman, McGraw-Hill, New York, 1963, pp. 207-216.

7. Kochen, M., "Adaptive Mechanisms in Digital 'Concept' Processing," _Discrete Adaptive Processes—Symposium and Panel Discussion_, AIEE, New York, 1962, pp. 50-58.

8. Kochen, M., "Some Problems in Information Science with Emphasis on Adaptation to Use Through Man-Machine Interaction," Vol. 1 & 2, IBM, Thomas J. Watson Research Center, Yorktown Heights, N.Y., April 1964.

9. Levien, R.E. and M.E. Maron, _Relational Data File: A Tool for Mechanized Inference Execution and Data Retrieval_, The RAND Corp., RM-4793-PR, December 1965.

10. Levien, R.E. and M.E. Maron, _A Computer System for Inference Execution and Data Retrieval_, The RAND Corp., RM-5085-PR, September 1966.

11. Lindsay, R.K., "Inferential Memory as the Basis of Machines Which Understand Natural Language," _Computers and Thought_, ed. E.A. Feigenbaum and J. Feldman, McGraw-Hill, New York, 1963, pp. 217-233.

12. Maron, M.E., _Relational Data File I: Design Philosophy_, The RAND Corp., P-3408, July 1966.

REFERENCES (cont'd)

13.   McIlroy. M.D., "Using SAP Macro Instructions to Manip-
       ulate Symbolic Expression," Computer Laboratory
       Memo, Bell Telephone Laboratories, Murray Hill,
       N.J., 1960.

14.   Mooers, C.N., "TRAC, A Procedure-Describing Language
       for the Reactive Typewriter," Comm. ACM., Vol. 9,
       No.3, March 1966, pp. 215-219.

15.   Mooers, C.N., and L.P. Deutsch, "TRAC, A Text Handling
       Language," Proc. ACM National Conference, Cleveland,
       August 1965, pp. 229-246.

16.   Newell, A. (ed.), Information Processing Language - V
       Manual, The RAND Corp., Prentice-Hall, Englewood
       Cliffs, N.J., 1961.

17.   Roberts, L.G., "Graphical Communications and Control
       Languages," Second Congress on Information System
       Sciences, Spartan Books, Baltimore, Md., 1964.

18.   Simmons, R.F., "Answering English Questions by Computer,
       A Survey," Comm. ACM, Vol. 8, No. 1, January 1965,
       pp. 53-70.

19.   Strachey, C., "A General Purpose Macrogenerator," Computer
       Journal, Vol. 8, No. 3, 1966.

20.   Sutherland, I.E., "Sketchpad, A Man-Machine Graphical
       Communication System," Proc. AFIPS 1963 SJCC, Spartan
       Books, Baltimore, Md., pp. 329-346.

21.   Weizenbaum, J., "Symmetric List Processor," Comm. ACM,
       Vol. 6, No. 9, September 1963, pp. 524-544.

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
THE UNIVERSITY OF MICHIGAN CONCOMP PROJECT	Unclassified
	2b. GROUP

3. REPORT TITLE

TRAMP: A RELATIONAL MEMORY WITH AN ASSOCIATIVE BASE

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Technical Report

5. AUTHOR(S) (First name, middle initial, last name)

William Ash and Edgar Sibley

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
May 1968	80	21

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
DA-49-083 OSA-3050	Technical Report 5
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DDC.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Advanced Research Projects Agency

13. ABSTRACT

This report describes the theory and implementation of an experimental language called TRAMP, which is a software simulation of a content-addressable memory. The system consists of an associative data structure embedded in an interpretive language, allowing great flexibility and strong recursive power. The system has further been extended with a logical inference capability by superimposing a relational structure over the associative memory. The resulting language has already proved to be extremely powerful in several applications, and can be termed a language for developing question-answering and interactive communication systems.

This report discusses the theory and design considerations, details of machine implementation, and details of operation with examples.

DD FORM 1473, 1 NOV 65

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computer Softwear Simulated Associative Memory						
Data-Structure						
Relations						
Recursive Associative Processor						
Question-Answering Languages						
Hash Coding						